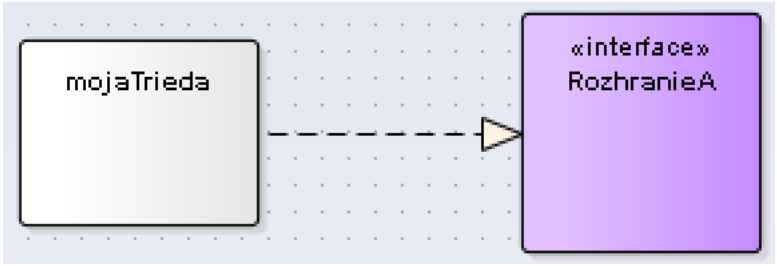
Rozhranie/implementácia (opakovanie)

- Rozhranie entity – špecifikuje čo nejaká entita vie/sľubuje, zhŕňa to, čo by mal okolitý svet o danej entite na jej použitie vedieť
- Implementácia – ma na starosti to, aby daná entita robila to, čo nasľubovala

Rozhranie (z časti opakovanie)

Novy dátový typ „rozhranie“ („interface“):

- posúva koncept abstraktnej triedy ďalej ...
- len deklaruje rozhranie
- nemá implementáciu (nemôže definovať!)
- môže obsahovať atribúty, tie sú však „public“, „static“ a „final“, t.j. jedná sa o konštanty.



- inštancuje sa pomocou objektov tried, ktoré dané rozhranie implementujú
  - inštancia rozhrania = inštancia triedy (objekt), ktorá dané rozhranie implementuje
- keďže rozhrania nemajú žiadnu implementáciu nehrozia problémy pri násobnom dedení a triedy môžu mať teda ľubovoľný počet rodičov – interfejsov bez spôsobovania zmätku
  - keď je treba viacnásobnú dedičnosť → vhodné použitie rozhrania!
- Rozhranie predstavuje niečo ako „protokol“ medzi triedami
- Rozhranie sa dedí

```
interface RozhranieA {
...
}
class MojaTrieda implements RozhranieA {
...
}
```

- Kedy použiť rozhranie a kedy abstraktnú triedu?
  - Keď tvoríte základnú triedu – **skúste najprv rozhranie**
  - ak treba mať definície metód, alebo atribúty … ok, zľaviť na abstraktnú triedu …
  - Ak má mať všetky definície … ok, zmeniť na konkrétnu triedu

Príklad: [4a: Akčná postava + Superman]

Rozhranie a udržateľnosť kódu

- Čo ak rozhranie chceme rozširovať?
- Ak do neho pridáme priamo požiadavky na nové metódy potom ostanú všetky triedy, ktoré rozhranie implementovali nefunkčné
- Riešenie?

```
public interface ViemRobitNiecoV1 {
   void urobAkciuX(int i);
}
```

```
public interface ViemRobitNiecoV2 extends ViemRobitNiecoV1{
   void urobAkciuY(int i, int j);
}
```

- Teraz sa každý môže rozhodnúť či bude implementovať pôvodné rozhranie, alebo prejde na nové

Rozhranie ako enum ...

- Pretože každý atribút, ktorý je obsiahnutý v rozhraní je automaticky static+final, je rozhranie bežným nástrojom na vytváranie skupín konštánt (niečo ako ENUM z C, C++). Napr:

```
public interface Mesiace {
     int
       JANUAR = 1, FEBRUAR = 2, MAREC = 3,
       APRIL = 4, MAJ = 5, JUN = 6, JUL = 7,
       AUGUST = 8, SEPTEMBER = 9, OKTOBER = 10,
       NOVEMBER = 11, DECEMBER = 12;
}
```

Enum

- Java má aj “enum“ - je to špeciálna trieda, ktorá premennej umožňuje nadobúdať hodnotu jednej z preddefinovaných konštánt
  - Táto trieda môže obsahovať atribúty a metody
  - Kompiler
    - hodnoty uloží to poľa
    - pridá ďalšie špeciálne metódy
- Takéto správanie “enum”u si vieme doprogramovať aj sami vo vlastnej réžii (spravíme si špeciálnu triedu) … ale pomocou “enum” to urobí JAVA za nás …
  - a nam stačí doprogramovať iba to, čo chceme ...
- enum môžeme používať pri príkazoch switch/case, atď …

Enum príklad 1 – jednoduché konštanty [Príklad: myEnum1]

```
enum Mesiace {
    JAN, FEB, MAR, APR, MAJ, JUN, JUL, AUG, SEP, OKT, NOV, DEC
}
public class MyEnum {
    public static void main(String[] args) {
        // TODO code application logic here
        System.out.print("Mame " + Mesiace.values().length + " mesiacov:");
        for(mesiace m : Mesiace.values())
            System.out.print(" " + m);
        System.out.println("");
        for(int i=0;i<Mesiace.values().length;i++) {
            System.out.println(mesiace.values()[i]+ "("+(i+1)+")");
        }
    }
}
```

Enum príklad 2 – N objektov [Príklad: myEnum2]

```
enum Planeta {
    MERKUR (3.303e+23, 2.4397e6),
    VENUSA (4.869e+24, 6.0518e6),
    …
    NEPTUN (1.024e+26, 2.4746e7);
    public final double vaha;    // v kilogramoch
    public final double polomer; // v metroch
    Planeta(double vaha, double polomer) {
        this.vaha = vaha;
        this.polomer = polomer;
    }
    …
}
```

Generický typ triedy (generická trieda)

- umožňuje parametrizáciu pomocou rôznych tried, niečo ako „templates“ v C++
- Uvažujme situáciu (bez explicitnej parametrizácie), keď chceme mať nad rôznymi objektami nejakú jednotnú sadu funkcionality
- Môžeme si vytvoriť kontajner a použiť typ „Object“, do ktorého uložíme ľubovoľný objekt
  - riskujeme, že ak spravíme triviálne chyby, kompilátor ich neodhalí ...
  - [Príklad: Kontajner1]

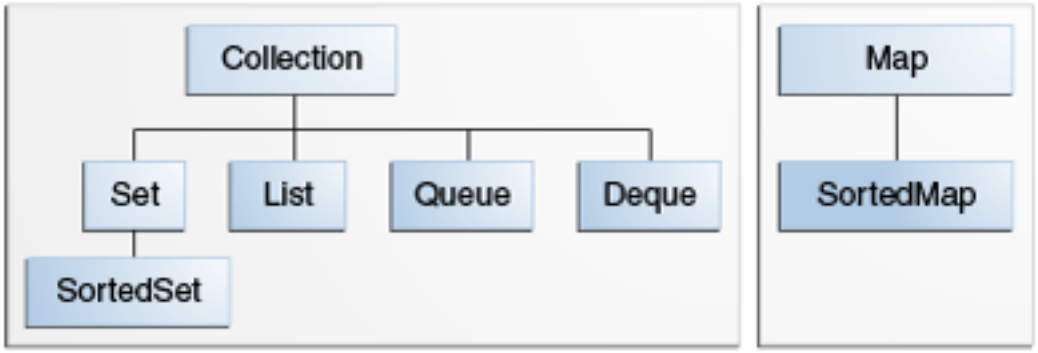
```
class Kontajner {
    final private Object polozka;   // polozka moze byť ľubovoľný objekt
    public Kontajner(Object objekt) { this.polozka = objekt; }
    public Object vrat() { return polozka; }
}
```

- Efektívne riešenie ?
  - vytvoríme deklaráciu triedy „generického typu“ s explicitným vyjadrením parametrizácie
  - „public class Trieda { ...} “ -> „public class Trieda<T> { ...}“
    - T – je typ s ktorým budeme pracovať
    - [Príklad: Kontajner2]

```
class Kontajner <T> {
    final private T polozka;   // polozka je typu T
    public Kontajner(T objekt) { this.polozka = objekt; }
    public T vrat() { return polozka; }
}
```

Využitie generického typu – Collections Framework

- Collection – kolekcia/kontajner – je objekt, ktorý vie zoskúpiť viacero elementov (E) do jednej do jednotky
- "Collections framework" je architektúra na reprezentáciu a prácu s kolekciami. Obsahuje
  - Rozhrania
  - Implementácie
  - Algoritmy
- Je to niečo ako Standard Template Library (STL) pri C++
- Rozhrania pri kolekciách tvoria hierarchiu (dá sa prekreslit v UML pomocou generalizácie):



- Deklarácia rozhrania kolekcií je napr. "public interface Collection<E>"
- Java neposkytuje žiadnu priamu implementáciu tohto rozhrania, namiesto toho sa používa:
  - Set – kolekcia, ktorá nemôže obsahovať duplicitné prvky
    - SortedSet – zoradená množina
  - List – kolekcia s poradím, ktorá môže obsahovať duplicity, má indexy
  - Queue – kolekcia s nejakým princípom zoraďovania, poskytuje metódy na vkladanie, výber, inšpekciu …
  - Map – object, ktorý mapuje páry kľúč-hodnota, pričom platí: Kľúče nemôžu byť duplicitné, každý kľúč sa mapuje na práve jednu hodnotu
    - SortedMap

ArrayList

- Príklad (už sme ho mali v jednom s minulých príkladov) :

```
List<String> myList = new ArrayList<String>();
Resp.
List<String> myList = new ArrayList<>();
Resp.
List<String> myList = new ArrayList();
```

- Trieda ArrayList implementuje rozhranie List (https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html):

java.util

Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

- Rozhranie Collection obsahuje metódy na vykonanie základných operácií ako:
  - **zistenie počtu prvkov, ... :** int size(), boolean isEmpty(), boolean contains(Object element)
  - **pridavanie/odoberanie prvkov ...:** boolean add(E element), boolean remove(Object element), a Iterator<E> iterator().
  - **metódy, ktoré pracujú s celými kolekciami:** boolean containsAll(Collection<E> c), boolean addAll(Collection<E> c), boolean removeAll(Collection<E> c), boolean retainAll(Collection<E> c), a void clear().

Prechádzanie kolekciami

[Prednaska04_objekty]

- „For each“

```
public class Student {
    private static final ArrayList<Student> gzoznam = new ArrayList();
    static void tlacZoznam() {
        for (Student stud : gzoznam)
            System.out.println("Student UID=" + stud.getUid() +
                               " Meno=" + stud.getMeno());
    }
```

- Súhrnné operátory (od JAVA 8 vyššie). Príklad implementácie metódy „tlacZoznam“:

```
gzoznam.stream().forEach(e -> System.out.println("MENO2: "+e.getMeno()));
```

- Iterátory (umožňuje počas prechádzania kolekciou aj vymazávať prvky):
  - rozhrania Iterátor a Iterable sú definované takto:

```
public interface Iterator<E> {
   boolean hasNext();
   E next();
   void remove(); //optional
}
```

```
public interface Iterable<E> {
   Iterator<E> iterator;
}
```

  - Príklad implementácie metódy „tlacZoznam“ (bez vymazávania ☺):

```
Iterator<Student> itr = gzoznam.iterator();
while(itr.hasNext()) {
   Student stud = itr.next();
   System.out.println("MENO3: "+stud.getMeno());
}
```

Iterátor verzus ListIterator

- Príklad: ListIterator<Student> litr = gzoznam.listIterator();
  - Umožňuje iterovať smerom naspäť (litr.hasPrevious(), litr.previous())
  - Umožňuje modifikovať zoznam (litr.set ...)

HashSet

- Implementuje rozhranie „Set“

```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

```
static final String[] S_IN={"jeden", "dva",
                     "jeden", "tri", "styri",
                     "dva", "sedem", "jeden"};
public static void main(String[] args) {
    Set<String> s = new HashSet();
    for (String a : S_IN)
        if (!s.add(a))
            System.out.println("Nájdená duplicita pri slove " + a);
    System.out.println("Počet jedinečných slov: " + s.size() + " - " + s);
}
```

HashSet – hromadné operácie

- Hromadné operácie sú napr: zjednotenie množín, prienik množín, rozdiel množín.
- Použime napr. rozdiel množín
  - set1=set1-set2
  - implementácia: set1.removeAll(set2)

Príklad:

```
static String[] sin={"jeden", "dva",
                     "jeden", "tri", "styri",
                     "dva", "sedem", "jeden"};
public static void main(String[] args) {
   Set<String> jedinecne = new HashSet<String>();
   Set<String> duplicitne= new HashSet<String>();

   for (String a : sin)
      if (!jedinecne.add(a)) duplicitne.add(a);

   // Deštruktívny rozdiel množín
   jedinecne.removeAll(duplicitne);

   System.out.println("Jedinečné slová:  " + jedinecne);
   System.out.println("Duplicitné slová: " + duplicitne);
```

Výstup:

Jedinečné slová: [styri, sedem, tri]

Duplicitné slová: [jeden, dva]

HashMap & LinkedHashMap

- Implementujú rozhranie „Map“
- LinkedHashMap naviac zachováva poradie
  - K - Kľúč (Key)
  - V - Hodnota (Value)

```
public class HashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
```

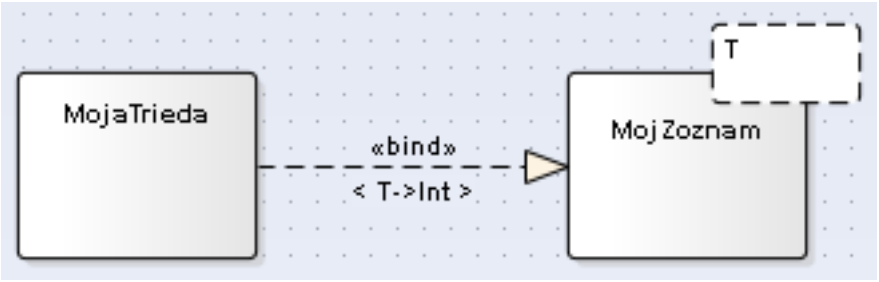
Príklad:

```
public static void main(String[] args) {
        LinkedHashMap<String, Integer> hm = new LinkedHashMap();
        hm.put("jano", 10);
        hm.put("emil", 5);
        hm.put("jozo", 40);
        hm.put("jozo", 50);
        hm.putIfAbsent("jozo", 90);
        hm.putIfAbsent("karol", 100);
        System.out.println(hm.keySet());
        for(String key : hm.keySet()) {
             System.out.println("KEY="+key+" VALUE="+hm.get(key));
        }
    }
```

Výstup:

[jano, emil, jozo, karol]
KEY=jano VALUE=10
KEY=emil VALUE=5
KEY=jozo VALUE=50
KEY=karol VALUE=100

Ako sa generické triedy značia pomocou UML?



Príklad: generická trieda s parametrom T, používa ju MojaTrieda.

```
class MojZoznam <T> {
    final private T polozka;   // polozka je typu T
. . .
}
```

```
class MojaTrieda {
    final private MojZoznam<Int> zoznam = new MojZoznam();
. . .
}
```

Doteraz sme sem-tam používali výnimky – exceptions ... čo to je?

- Výnimka je udalosť, ktorá nastane počas vykonávania programu a ktorá naruší normálny tok príkazov programu
  - podrobnosti o výnimke sú v objekte výnimky
  - odovzdanie tohto objektu do bežiaceho systému za volá vyvolanie výnimky (throwing an exception)
  - bežiaci systém sa snaží nájsť blok kódu, ktorý výnimku obslúži – obsluha výnimky (exception handler)
  - ak typ výnimky súhlasí s typom, ktorý môže byť spracované obsluhou výnimky dochádza k zachyteniu výnimky (catch the exception)
  - ak bežiaci systém prehľadal metódy zo zásobníka a nenašiel vhodnú obsluhu výnimky, je bežiaci systém ukončený.
- Tri druhy výnimiek
  - kontrolovaná výnimka (napr. pri chybnom zadaní názvu súboru na deserializáciu objektu)
  - chyba (napr. zlyhá HW pri čítaní súboru)
  - výnimka bežiaceho programu (logické chyby, ...)
- try {...} catch {...} – ak v bloku „try“ nastane výnimka tak v bloku „catch“ je šanca na obsluhu výnimky
  - blok „finally“ – sa vykoná vždy, hneď po try/catch

```
try {
 ...
} catch (FileNotFoundException e) {
    System.err.println("Súbor nebol nájdený (FileNotFoundException): "
                       + e.getMessage());
...
} finally {
...
}
```

Pravidlá pri programovaní – „best practices“

- V triedach používať **„gettery“, „settery“** a **privátne atribúty**.
- Funkcie/metódy by **nemali vracať polia** (namiesto toho by mali vracať zoznamy – **Collection**)
  - namiesto null treba vracať prázdnu Collection
- **nezakrývať mená atribútov lokálnymi premennými** (ani nepoužívať podobné názvy)
  - dá sa explicitne odlišovať pomocou „this“ ale v prípade omylu/vynechania prefixu vznikajú ťažko odhaliteľné chyby
  - napr. na atrubúty používať prefix „m“, napr. „mCounter“ a lokálna premenná môže byť „counter“
- **vyhýbajte sa importom celých balíkov**, importujte len špecifické triedy
- **Veľkosť metód**
  - optimálne **10 riadkov kódu**, resp. to, čo sa **zmestí na obrazovku**
  - používajte **30 sekundové pravidlo** – metóda má byť tak veľká a tak napísaná aby iný programátor za 30 sekúnd pochopil, čo presne metóda robí
- vyhýbajte sa viac ako **2-3 vnoreným cyklom**
- vyhýbajte sa **zložitým logickým výrazom** (kto to má za 3 sekúnd analyzovať)?
- Pri porovnávaní **preferujte konštanty na ľavej strane**“ if (1 == nieco) {...}
  - minimálne v C++ by sa to mohlo vypomstiť v pripade preklepu, v jave to je bezpecnejšie (nekompatibilné typy)
  - resp. treba používať if (“”.equals(mojString)) {...}, lebo mojString by mohol byť null ...
- na začiatku spraviť radšej **dobrý objektovo orientovaný návrh** (ktorý možno bude pomalší ale bude fungovať. Až následne vytipované časti zrýchlovať za cenu straty čistoty návrhu.

Návrhové vzory

- V doterajších príkladoch sa vyskytovali viaceré návrhové vzory
- Prečo je dobré poznať návrhové vzory?
  - Ušetria čas
  - Zvyšujú kvalitu
  - Zlepšujú porozumenie pre vývojárov opravujúcich alebo rozširujúcich zdrojový kód
- História
  - Návrhové vzory získali popularitu po uvedení knihy „Elements of Reusable Object-Oriented Software“ v 1994 skupinou "Gang of Four" (GoF): Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides na konferencii OOPSLA v Portlande
- Čo je to návrhový vzor?
  - Návrhový vzor je popis alebo šablóna ako riešiť nejaký typ softvérového problému bez konkrétne špecifikácie aplikačných tried a objektov. Návrhový vzorec nie je hotový návrh, ale môže byť jednoducho transformovaný do kódu.
  - Doménou návrhových vzorcov sú abstraktné triedy, objekty a relácie medzi nimi .
  - Nie všetky softvérové vzory sú návrhové vzory. Napríklad, algoritmy riešia výpočtové problémy a nie návrhové problémy.