Programovanie, algoritmy, zložitosť / ÚINF/PAZ1c

PAZ1C

Róbert Novotný
robert.novotny@upjs.sk

13. 10. 2010

Zapúzdrenie

- *„proces „zaškatuľkovania" elementov abstrakcie, ktoré tvoria jej štruktúru a správanie. Účelom zapúzdrenia je oddeliť rozhranie s kontraktom abstrakcie od jej implementácie"*

—Grady Booch, *Object-Oriented Analysis and Design with Applications*, 2007

Štruktúra a správanie

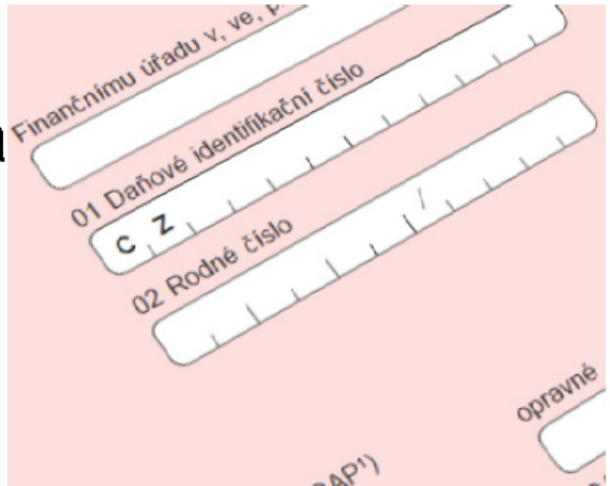
- *štruktúra* = **stav** = inštančné premenné
- *správanie* = **schopnosti** = metódy
- *„zaškatuľkovaný" element abstrakcie* = **trieda**
- *kontrakt* = **hlavičky metód** = formálna syntax pre to, čo od triedy očakávame
- *implementácia* = **kód** v metódach

Stav a schopnosti

- aký **stav** a **schopnosti** má mať trieda?
- úvahy nad kontraktom by mali mať prednosť pred úvahami nad implementáciou
- návrh stavu často záleží od očakávaných schopností

Príklad s rodným číslom

- aký **stav** a **schopnosti** má mať RČ?
- pýtajme sa naopak: najprv zistime schopnosti, od nich odvoďme stav
- aké **schopnosti** očakávame od rodného čísla?
  - zistiť deň, mesiac a rok narodenia
  - zisti, či je korektné
  - zisti, či je majiteľ muž alebo žena
  - daj reťazcovú reprezentáciu
    - s lomkou i bez



Príklad s rodným číslom

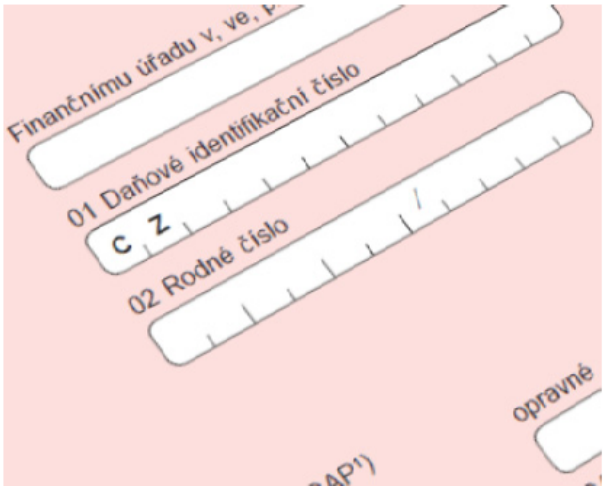

```
class RodneCislo {
  int getRok()
  int getMesiac()
  int getDen()
  boolean jeMuzske()
  boolean jeValidne()
  String toString()
  String toStringBezLomky()
}
```

Rodné číslo – ako reprezentovať?

- ako interne reprezentovať rodné číslo?
- možnosť 1: jeden String
  - nevýhoda: ak chceme vypisovať s lomkou / bez lomky, musíme manipulovať so Stringami
  - nevýhoda: validácia = deliteľnosť štyroma
- možnosť 2: štyri integerové premenné: deň, mesiac, rok, prípona
  - nevýhoda: treba vysekávať zo Stringovej reprezentácie, nezabudnúť na pripočítavanie 5 u žien
  - nevýhoda: pozor na to, že prípona môže začínať nulou, to isté jednotlivé zložky

Rodné číslo – ako reprezentovať?

- ako interne reprezentovať rodné číslo?
- možnosť 3: jeden int
  - nevýhoda: treba vysekávať zo Stringovej reprezentácie, nezabudnúť na pripočítavanie 5 u žien
  - pozor na to, že rodné číslo môže začínať nulou (deti narodené 2000-2009)



Akú reprezentáciu zvoliť?

- Používateľa triedy nezaujíma, akú reprezentáciu zvolíme, dôležité je, že jeho metódy robia to, čo sa od nich čaká
- vonkajší pohľad na triedu ustanovuje **kontrakt**
  - záruky správania a predpoklady na ktoré sa môže používateľ triedy spoliehať
  - očakávania používateľa, ktoré musí trieda naplniť
  - kontrakt zároveň určuje zodpovednosti triedy

Akú reprezentáciu zvoliť?

- **Kontrakt** je definovaný hlavičkami verejných metód
  - parametre a ich typy
  - návratové hodnoty
- Správanie metód je záležitosťou implementácie
  - trieda sa správa ako čierna skrinka

Príklad s rodným číslom

- v triede RodnéhoČísla sme definovali kontrakt
- implementácia spočíva v návrhu inštančných premenných a v interných algoritmoch, ktoré využívajú stav

```
class RodneCislo {
  int getRok()
  int getMesiac()
  int getDen()
  boolean jeMuzske()
  boolean jeValidne()
  String toString()
  String toStringBezLomky()
}
```

Príklad s rodným číslom

- zvoľme si reprezentáciu jedným Stringom
- jedna inštančná premenná
- kód v metódach bude závisieť na reprezentácii
- `jeMuzske()` – zistí, či sa 5. miesto začína nulou alebo jednotkou
- ...

```
class RodneCislo {
  private String rč;
  RodneCislo(String rc);

  int getRok()
  int getMesiac()
  int getDen()
  boolean jeMuzske()
  boolean jeValidne()
  String toString()
  String toStringBezLomky()
}
```

Príklad s rodným číslom

- Použitie triedy:

```
RodnéČíslo rč = new RodnéČíslo("751212/8823");
if(rč.jeValidné()) {
  System.out.println("Osoba sa narodila"
    + " v roku " + rč.getRok());
}
```

Zapúzdrenie a jeho výhody

- Vďaka zapúzdreniu môžeme v prípade potreby zmeniť internú implementáciu
- Ak dodržíme kontrakt, používateľ si nič nevšimne
- Čierna skrinka!

Príklad s rodným číslom

- Zvoľme štyri integerové premenné:

```
class RodneCislo {
  private int deň;
  private int mesiac;
  private int rok;
  private int prípona;
  ...
}
```

- Ak dodržíme kontrakt, použitie kódu je rovnaké.

```
RodnéČíslo rč = new RodnéČíslo("751212/8823");
if(rč.jeValidné()) {
  System.out.println("Osoba sa narodila"
    + " v roku " + rč.getRok());
}
```

Kontrakty ako spôsob návrhu

- **kontrakt** je dohoda medzi klientom a triedou
- „ak klient splní záväzky, trieda splní očakávania klienta"
- kontrakt hovorí **„ČO"** trieda vykoná
  - implementácia triedy hovorí "AKO" sa to dosiahne
  - to však klienta nezaujíma
  - trieda je čierna skrinka

Interfejs = vyjadrenie kontraktu

- V Jave možno vyjadriť kontrakt pomocou metód interfejsu
- **Interfejs** = zoznam metód = zoznam schopností triedy
- ak trieda implementuje interfejs, hovorí, že dokáže poskytnúť danú schopnosť
- v Jave: kľúčové slovo **interface**

Príklad: dátová štruktúra Zásobník

- zásobník (***stack***) je zoznam, kde položky možno len vkladať na vrchol a odoberať z vrchola
  - **LIFO** – *last in-first out* (posledný dnu-prvý von)
- čo má poskytovať zásobník?
  - **push()** – vlož na vrchol zásobníka
  - **pop()** – vyber z vrchola
  - **isEmpty()** – je zásobník prázdny?
  - **size()** – počet prvkov v zásobníku

Interface k zásobníku


```
public interface Stack {
   void push(Object o);
   Object pop();
   boolean isEmpty();
   int size();
}
```

- interface udáva len hlavičky metód
- žiadne kučeravé zátvorky
- filozoficky zodpovedá triede
  - všetky metódy sú verejné (netreba písať public, ale robí sa to)

Interface k zásobníku


```
public interface Stack {
  void push(Object o);
  Object pop();
  boolean isEmpty();
  int size();
}
```

```
Stack zásobník = new Stack();
```

- nemožno vytvoriť inštanciu!
- veď zásobník nemá kód v metódach
- teda nie je jasné, **AKO** sa majú metódy vykonať
- to povie až trieda, ktorá bude tento interfejs **implementovať**

Implementovanie interfejsu

PAZ1C

```
public class ListBasedStack implements Stack {
  private ArrayList data = new ArrayList();
  public void push(Object o) { data.add(o); }
  public Object pop() {
    if(data.isEmpty()) {
      return null;
    }
   return data.get(data.size() – 1);
  }
  public boolean isEmpty() { return data.isEmpty(); }
  public int size() { return data.size(); };
}
```

Použitie v kóde

- naša trieda **ListBasedStack** **implementuje** interfejs **Stack**
- hovorí „**ako**" sa metódy budú správať
- použitie u klienta:

```
Stack zásobník = new ListBasedStack();
```

- dátový typ vľavo = **ČO** = aký kontrakt chceme používať
- dátový typ vpravo: = **AKO** = ktorá trieda nám ho poskytne

Zásada pre používanie interfejsov

Programujte vzhľadom k interfejsom, nie k ich implementáciám

Program to an interface, not to an implementation!

—*Gang of Four (Gamma, Helm, Johnson, Vlissides),* **Design Patterns**

Programujte vzhľadom k rozhraniam!

- vždy, keď je to možné, používajte pre premenné interfejsy, nie konkrétne implementácie
  - v parametroch metód
  - v návratových hodnotách
  - v lokálnych premenných na ľavej strane
- umožníte tým dodržať zásadu čiernej skrinky

Interfejsy a implementácie


```
public class SprávcaŠtudentov {
   private List<String> študenti = new ArrayList<String>();
   public List<String> getŠtudenti() {
      return študenti;
   }
}
```

- **interfejsy** v type inštančnej premennej
  - klienta zaujíma, že dostane zoznam študentov, nemusí ho zaujímať, ako je tento zoznam implementovaný (tu: **ArrayList** = zoznam nad poľom)
- **implementácie** pri konštruovaní

Akú to má výhodu?

- ak všade používame implementácie a rozhodneme sa nahradiť ich, máme problém
  - všade používajme **ArrayListy**
  - lenže zrazu sa ukáže, že ich chceme nahradiť **CopyOnWriteArrayListom**, ktorý podporuje súčasný prístup z viacerých vlákien
- zbesile nahrádzanie v kóde
- **strašné** narušenie kompatibility
  - kód, ktorý používa naše triedy sa musí prepísať
  - „pokazia" sa oddedené triedy

Reálny príklad (pozitívny)

- trieda **JList**: ovládací prvok pre zobrazenie zoznamov

```
public setModel(ListModel model)…
```

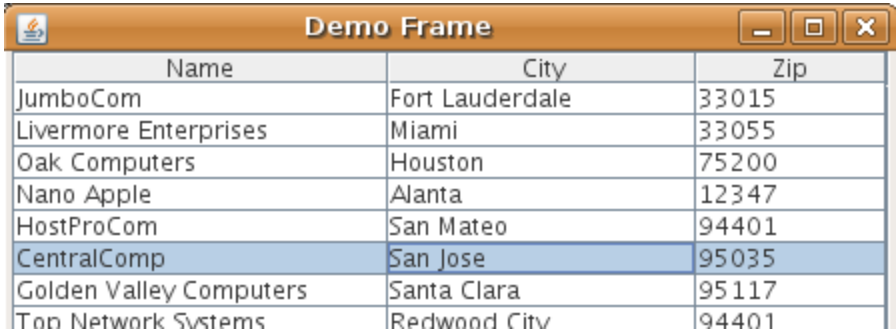
- javax.swing.**ListModel** – model (nositeľ) dát zobrazených v zozname
- môžeme dodať štandardný model **DefaultListModel**
  - **ListModel** simulujúci zoznam
- alebo môžeme dodať ľubovoľnú inú vlastnú implementáciu
  - ťaháme dáta z databázy, súborového systému…

Reálny príklad (aj majster tesár...)

- trieda **JTable**: ovládací prvok pre zobrazenie tabuľkových dát

```
public JTable(Vector rowData, Vector columnNames)…
```

- java.util.**Vector** – trieda pre zoznamy (sťa ArrayList)
- uvedenie implementačnej triedy robí z tohto konštruktora **nepoužiteľnú** (= zbytočnú) vec

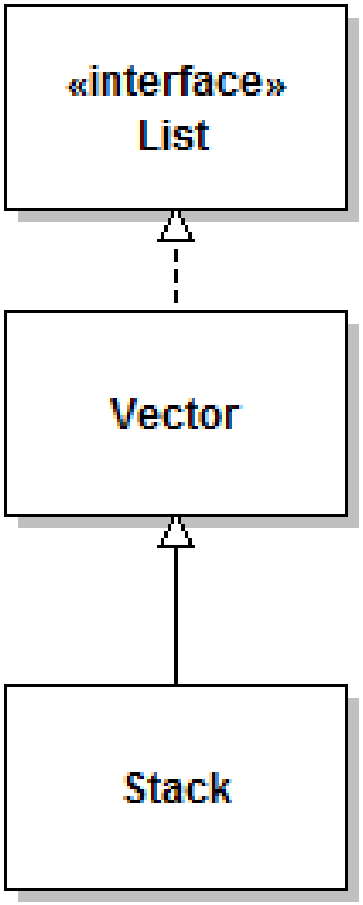


Kdepak udělali soudruzi zo Sunu chybu?

PAZ1C

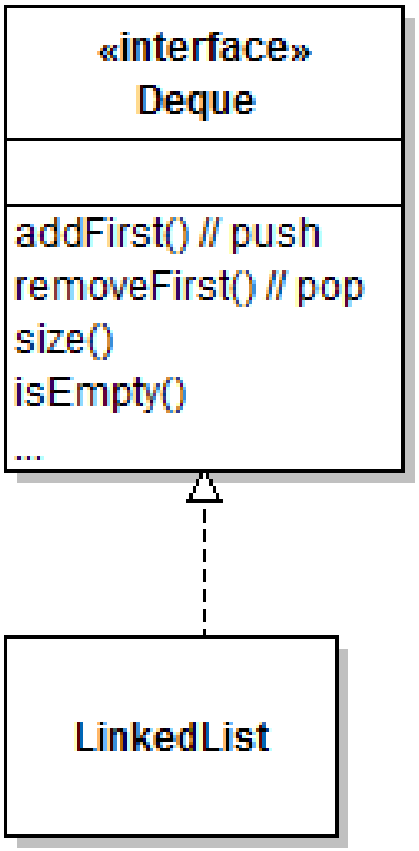
- dávno predávno (Java 1.0) existovala jediná trieda pre zoznamy: **java.util.Vector**
- Java 1.2 (1998): Joshua Bloch prekopal triedy pre kolekcie
- kopa interfejsov (**List**, **Set**, **Map**)...
- **Vector** sa stal zastaralým (namiesto neho **ArrayList**)
  - ešteže ArrayList implements List, Vector implements List
- žiaľ, konštruktor v **JTable** je nepoužiteľný, lebo očakáva implementáciu a nie interfejs
- ospravedlnenie: vtedy sa mnoho vecí nevedelo predvídať

Iný príklad utnutého majstra



- **Stack extends Vector**
- narúša to pravidlo „je"
  - „zásobník nie je vektorom založeným na poli"
- narúša to Liskovovej substitučný princíp
  - zásobník je zoznam, do ktorého možno vkladať len na koniec

Oprava chýb minulosti



- v JDK 6 – **Deque**
- interfejs pre „double-ended queue"
  - rad s dvoma koncami
- existuje viacero implementácií
  - **LinkedList**
  - **ArrayDeque**

Zásobník v čiernej skrinke spojového zoznamu


```
public boolean testujZatvorky(String vyraz) {
  Stack<Character> zasobnik = new Stack<Character>();
  for (int i = 0; i < vyraz.length(); i++) {
    if(vyraz.charAt(i) == '(') zasobnik.push('(');
    if(vyraz.charAt(i) == '[') zasobnik.push('[');
    try {
      if((vyraz.charAt(i) == ')')
        && (zasobnik.pop() != '(')) return false;
      if((vyraz.charAt(i) == ']')
        && (zasobnik.pop() != '[')) return false;
    } catch (EmptyStackException e) {
      return false;
    }
  }
  return zasobnik.empty();
}
```

Zásobník v čiernej skrinke spojového zoznamu


```
public boolean testujZatvorky(String vyraz) {
  Map<Character, Character> mapaZnakov
    = new HashMap<Character, Character>();
  mapaZnakov.put(']', '[');
  mapaZnakov.put(')', '(');
  Deque<Character> zasobnik = new LinkedList<Character>();
  for (char znak : vyraz.toCharArray()) {
    if(mapaZnakov.values().contains(znak)) {
      zasobnik.push(znak);
    } else {
      Character z1 = mapaZnakov.get(znak);
      Character z2 = zasobnik.poll();

      if(z1 == null || z2 == null || !z1.equals(z)) {
        return false;
      }
    }
  }
  return zasobnik.isEmpty();
}
```

Voláme metódy na interfejse!

Interfejs ako schopnosť

- interfejs je vhodná forma na špecifikáciu kontraktu
  - **„ČO"** chceme
- zároveň určuje schopnosti triedy, ktorá ho implementuje
- interfejs možno použiť na špecifikáciu **roly** triedy

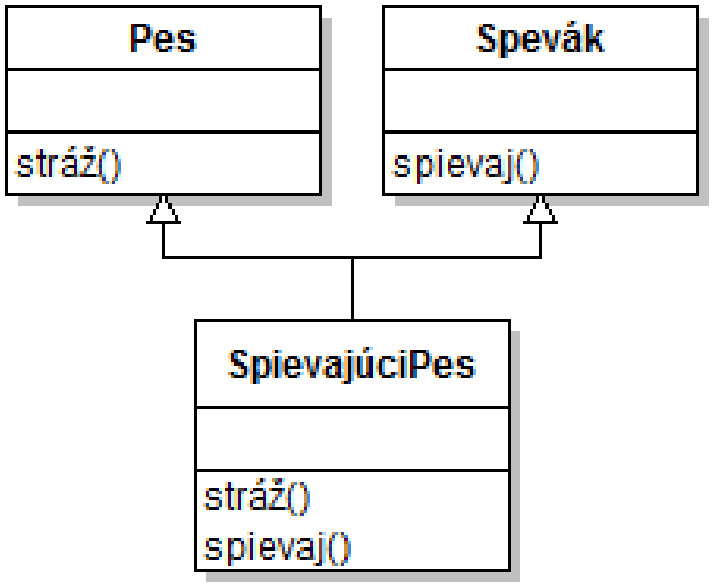
Schopnosť dediť od viacerých tried

- Príklad: máme triedu **Pes** a triedu **Spevák**. Chceme **spievajúceho psa**
  - **Pes –** trieda poskytujúca schopnosť strážiť
  - **Spevák** – trieda poskytujúca schopnosť lúdiť tóny

```
class Pes {
   void stráž() {
      …
   }
}
```

```
class Spevák {
   void spievaj() {
      …
   }
}
```

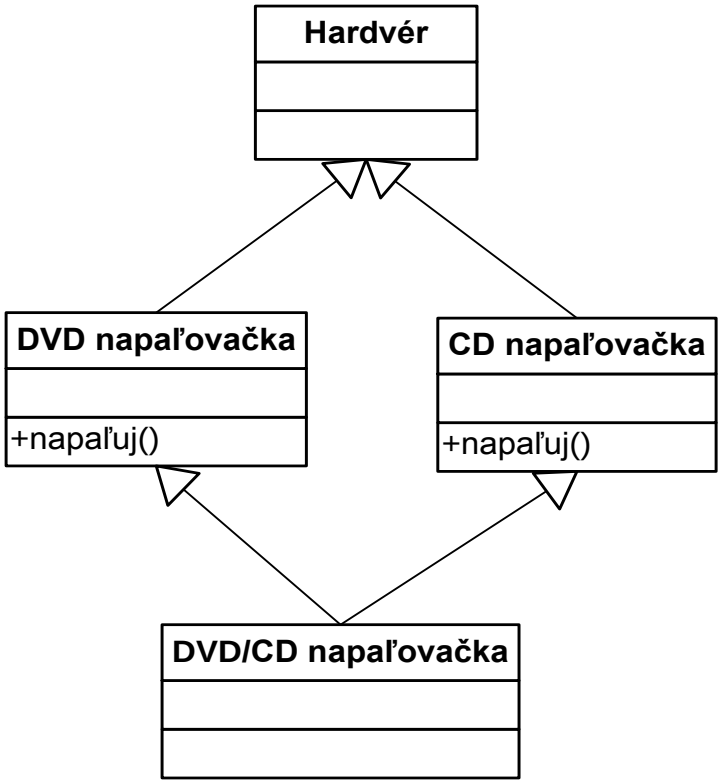
Viacnásobná dedičnosť



- viacnásobná dedičnosť však v Jave neexistuje!
- nemožno dediť od viacerých tried

```
class SpievajúciPes
   extends Pes, Spevák {
...
}
```

Viacnásobná dedičnosť bola explicitne zakázaná



- „Smrtiaci smaragd smrti" (*Deadly diamond of Death*)
- DVD/CD napaľovačka zdedí obe metódy na napaľovanie – ale ktorá sa má zavolať kedy?

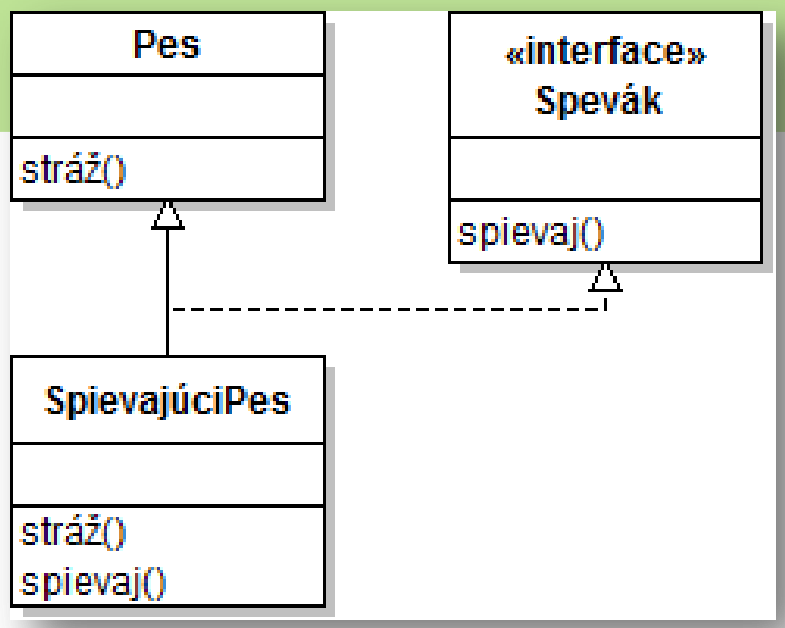
Nie je to však tragédia

- snaha o viacnásobnú dedičnosť často znamená snahu o triedu, ktorá napĺňa viacero rolí
- **spievajúci pes** – **pes**, ktorý dokáže naplniť rolu speváka
- roly vyjadríme interfejsom
- trieda naplní rolu tým, že implementuje interfejs

```
interface Spevák {
  void spievaj();
}
```

Implementácia interfejsu


```
public SpievajúciPes extends Pes implements Spevák {
   //táto metóda tu MUSÍ byť, prikazuje nám to interface
   public void spievaj() {
       System.out.println("Máááám rozpráávkovú búúúdu!");
   }
}
```



Trieda môže napĺňať viacero rolí


```
public ŠaľenyPes extends Pes
   implements Spevák, Comparable, Serializable, Cloneable
{
   //..
   //milión metód vyžadovaných od rozhraní
   //...
}
```

```
ŠaľenyPes einstein = new ŠaľenyPes();
Pes p = einstein;

Comparable c = new ŠaľenyPes();
Spevák spevák = new ŠaľenyPes();
```

- funguje dedičnosť aj polymorfizmus

Praktický príklad

- vezmime si typické operácie nad zoznamami
  - vykonaj nad každým prvkom operáciu
  - nájdi (jeden) prvok spĺňajú kritérium
  - nájdi všetky prvky spĺňajúce kritériá
  - prefiltruj zoznam

Metóda vráti podreťazec od počiatočného indexu po koncový (okrem neho).

Interfejsy môžu nahradiť lepenie kódu

- ako vyzerá algoritmus pre **each**?

1. bež v cykle cez zoznam
2. vezmi prvok
3. *aplikuj naň požadovanú operáciu*
   - ak rátame dvojnásobky, operácia zdvojnásobí prvok
   - ak rátame piate odmocniny, dodáme príslušnú operáciu

prvé dva kroky sa nikdy nemenia
tretí rok je premenlivý => **vyextrahujeme ho do interfejsu**

Prípad: each

- operáciu vieme reprezentovať interfejsom

```
interface Operácia {
  void vykonaj(int element);
}
```

```
public class OperácieSoZoznamami {
  public static void spracuj(List<Integer> list,
                             Operácia o) {
    for(int element : list) {
       o.vykonaj(element);
    }
  }
}
```

Prípad: each – vypíš dvojnásobky


```
public class VypíšDvojnásobky implements Operácia {
  public void vykonaj(int element) {
    System.out.println(element * 2);
  }
}
```

```
List<Integer> cisla = Arrays.asList(2, 4, 6);
Operácia o = new VypíšDvojnásobky();
OperácieSoZoznamami.spracuj(cisla, o)
```

- operáciu, ktorú vykonávame, určíme používaným interfejsom
- stačí vytvoriť triedu implementujúcu interfejs a dodať do nej kód
- metóde **spracuj()** dodáme zoznam a triedu s kódom, ktorý sa má vyvolať pri prechádzaní zoznamu

Interfejsy môžu nahradiť lepenie kódu

- návrhový vzor (design pattern!)
- algoritmus má viacero krokov
- niektoré sú nemenné
- niektoré sa menia a to za behu
- riadky, ktoré používať „maže a mení" vieme vložiť do metódy interfejsu
- v kóde algoritmu potom voláme príslušnú metódu interfejsu, za ktorou je konkrétna implementácia

Interfejsy môžu nahradiť lepenie kódu

- tento návrhový vzor sa používa kade-tade

```
File mp3Adresar = new File("D:/mp3");
File[] súbory = mp3Adresar.listFiles(FileFilter filter);
```

- **FileFilter** umožňuje vložiť inštanciu filtra, ktorý pre každého potomka adresára povie, či sa má vložiť do výsledného poľa.

```
interface FileFilter {
  boolean accept(File file);
}
```

Používateľ musí vytvoriť implementáciu tohto nterfejsu, vytvoriť jej inštanciu a hodiť ju do parametra.

Interfejsy môžu nahradiť lepenie kódu


```
public class Mp3FileFilter implements FileFilter {
  boolean accept(File file) {
     return file.getName().endsWith(".mp3")
  }
}
```

```
File mp3Adresar = new File("D:/mp3");
FileFilter filter = new Mp3FileFilter();
File[] súbory = mp3Adresar.listFiles(filter);
```

- v poli **súbory** sa zjavia len súbory končiace sa na **.mp3**