Polymorfismus, Modularita a Zapouzdření

OMO, LS 2014/2015

Typy

- Primitivní
  - logické hodnoty, znaky, čísla, adresy, …
- Složené
  - kartézský součin (součinové), např. **třídy**
  - disjunktní sjednocení (součtové), např. **union v C**
  - zobrazení (pole, funkce) (mocninné)

Typy jazyků

- Programovací jazyky můžeme dále rozdělit z hlediska typů do dvou skupin na:
  - **monomorfní** - každá hodnota nebo proměnná je **právě jednoho typu**
  - **polymorfní** - hodnoty a proměnné mohou mít **více typů**
    - **parametrický polymorfismus** - v typových výrazech se vyskytují typové proměnné (např. generika), za něž lze dosadit libovolný typ
    - **podtypový polymorfismus** - v typovém systému se zavede mezi typy relace <: a každá hodnota má kromě svého (nejmenšího) typu i všechny nadtypy

Princip subsumpce

- Nechť *a* je typu *A* a *A*, *B* jsou typy takové, že *A* je podtypem *B*, pak *a* je i hodnotou typu *B*.

$$\frac{a : A \qquad A <: B}{a : B}$$

Subsumpce příklad

```
interface A {
   void method();
}

class B implements A {
   void method() {...}
}

class C implements A {
   void method() {...}
}
```

```
B b = new B();
b.method();
A a = new C();
a.method();
a = b;
a.method();
b = a;//Chyba
```

Substituční princip (LSP)

- Nechť q(x) je vlastnost, která platí pro všechny objekty typu T. Pak q(y) by měla být platná pro objekty y, které jsou typu T', kde T' je podtypem T.

$$\frac{\forall\, x : T \;\; : q(x) \qquad T' <: T}{\forall y : T' \;\; : q(y)}$$

Co je polymorfismus?

- Obecně se polymorfismem rozumí **schopnost vyskytovat se ve více formách**. To jest, že jedna hodnota může mít **více typů**.
- Polymorfismus v Java
  - Schopnost reference chovat se dle skutečně **odkazovaného objektu**.
  - Výběr metody, která se volá probíhá za běhu programu podle toho na který objekt reference odkazuje.

Polymorfismus v Java

- V Java máme dostupný jak **podtypový polymorfismus**, tak **parametrický**.

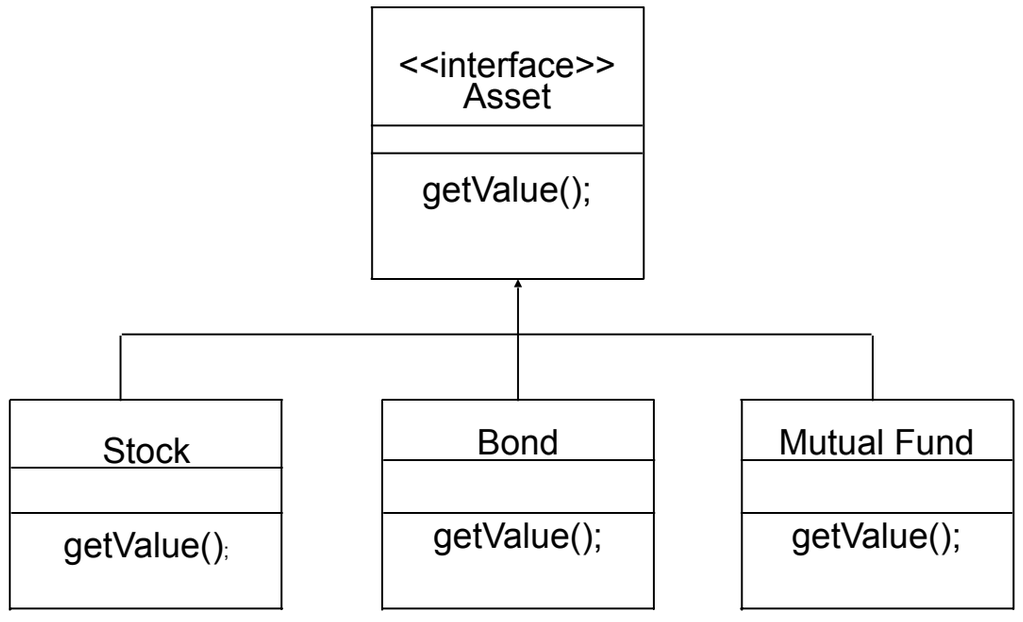
- **Podtypový polymorfismus** je realizován pomocí **rozhraní a dědičnosti tříd**.

- Parametrický polymorfismus je realizován **pomocí generic**.

- Podtypový polymorfismus vychází z principu **subsumpce**.

Polymorfismus

- Je to způsob jak skrýt více implementací za jedno rozhraní.
- Umožňuje aby stejná zpráva byla obsloužena různě v závislosti na konkrétním objektu.



Příklad

- Mějme abstraktní třídu *Shape,* polymorfismus umožňuje programátorovi definovat různé metody *area* ve zděděných třídách Circle, Rectangle, ….

- Zavolání metody *area* na referenci typu shape vrátí správný výsledek s ohledem na skutečně odkazovaný objekt.

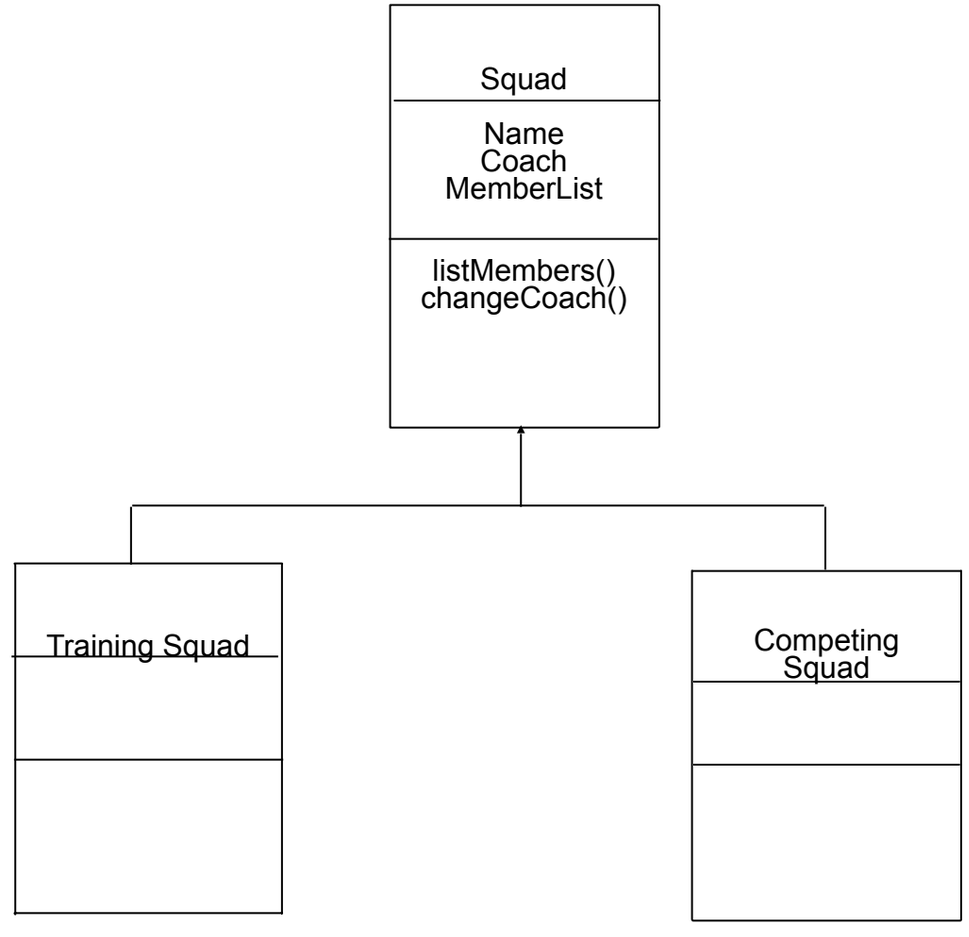
Generalizace

- Forma asociace, kde třída sdílí strukturu anebo chování jiných/jiné tříd/třídy.
- Definuje hierarchii.
  - **Jednoduchá dědičnost**
  - **Vícenásobná dědičnost**
- Je to vztah typu *kind of*.

Dědičnost

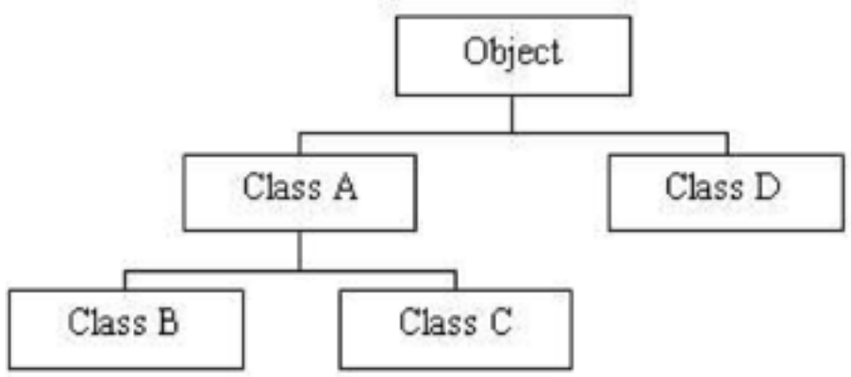
- Mechanismus jak více specifické třídy přejímají chování a strukturu obecnějších tříd.

- Třída dědí atributy, operace a vazby.



Dědičnost

- V Java jsou **všechny třídy zděděné z třídy** *Object*.
- V Java je k dispozici **jednoduchá dědičnost**.
- Hierarchie tříd se znázorňuje obvykle stromem



Dědičnost

- Výhoda dědičnosti v OOP - **Znovupoužitelnost**
  - **Chování (metody) definované v nadtřídě je dostupné ve všech podtřídách.**
  - Není nutné funkčnost metod definovat znovu.
  - Podtřídy mohou implementaci metod měnit a přidávat metody nové.

Dědičnost

- V Java používáme pro specifikaci dědičnosti klíčové slovo *extends* u třídy, která dědí.

```
public class Person {
      protected String name;
      protected String address;

      /**
       * Default constructor
       */
      public Person(){
            System.out.println("Inside Person:Constructor");
            name = ""; address = "";
      }
      . . . .
}

public class Student extends Person {
      public Student(){
            System.out.println("Inside Student:Constructor");
      }
      . . . .
}
```

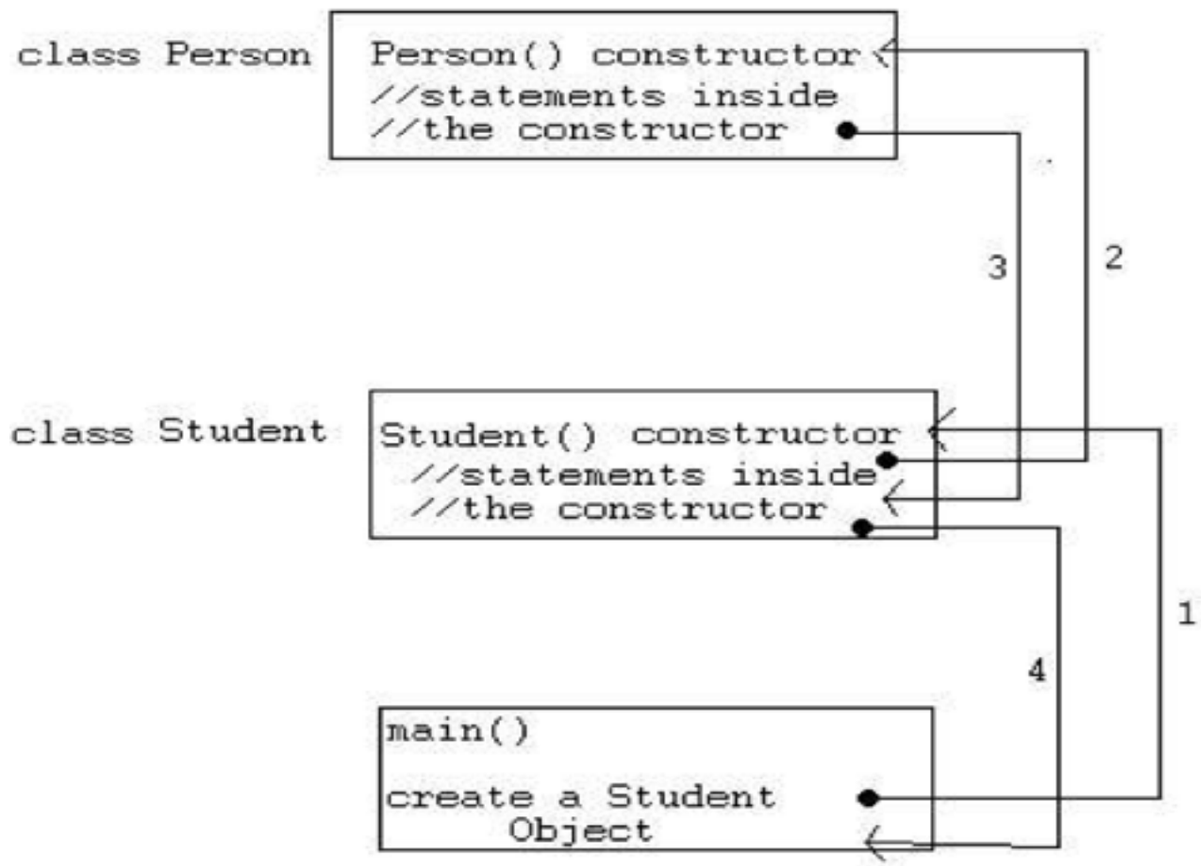
Dědičnost

Metoda main:

```
public static void main( String[] args ){
     Student anna = new Student();
}
```

Výstup programu:

```
Inside Person:Constructor
Inside Student:Constructor
```



Klíčové slovo “super”

- Podtřída může explicitně zavolat konstruktor nejbližší nadtřídy pomocí klíčového slova *super.*

```
public Student(){
     super( "SomeName", "SomeAddress" );
     System.out.println("Inside
Student:Constructor");
}
```

Klíčové slovo “super”

- Několik věcí na které je třeba myslet při používání klíčové slova super pro volání konstruktoru nadtřídy:
  - *super()* musí být **jako první příkaz konstruktoru**.
  - *super()* může být **použito pouze v konstruktoru**.
  - **Ve stejném konstruktoru nemůžeme použít *this()* a *super()* zároveň.**

Klíčové slovo “super”

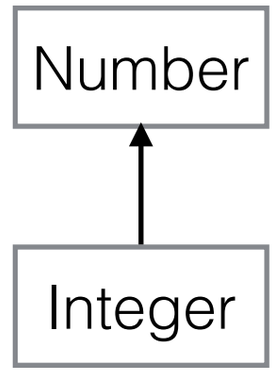
- Další použití je pokud chceme přistupovat ke členským proměnným nadtřídy.

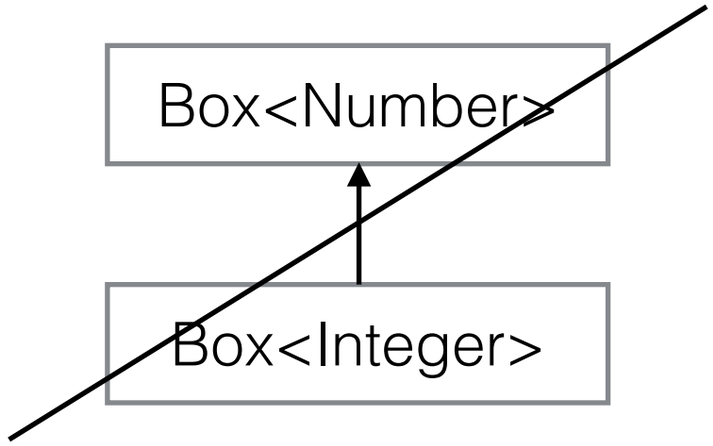
```
public Student() {
     super.name = “somename”;
     super.address = “some address”;
     super.s();
}
```

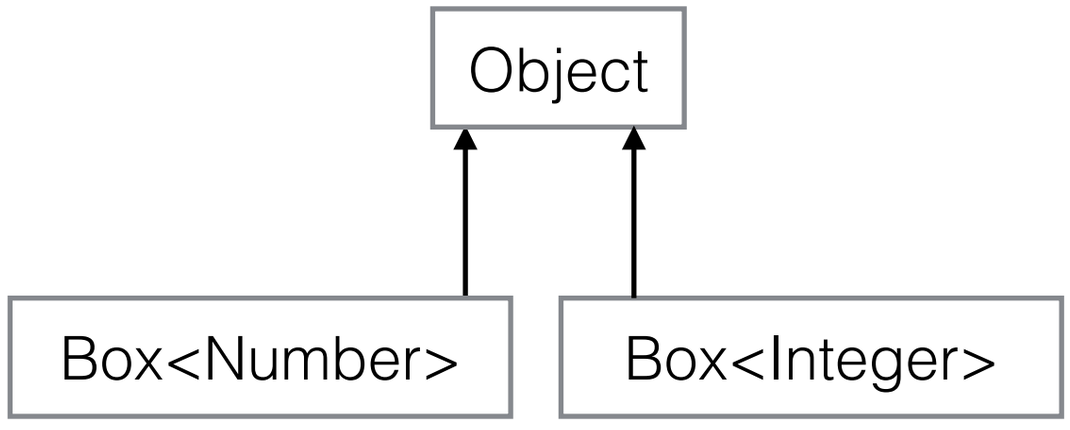
Generické typy

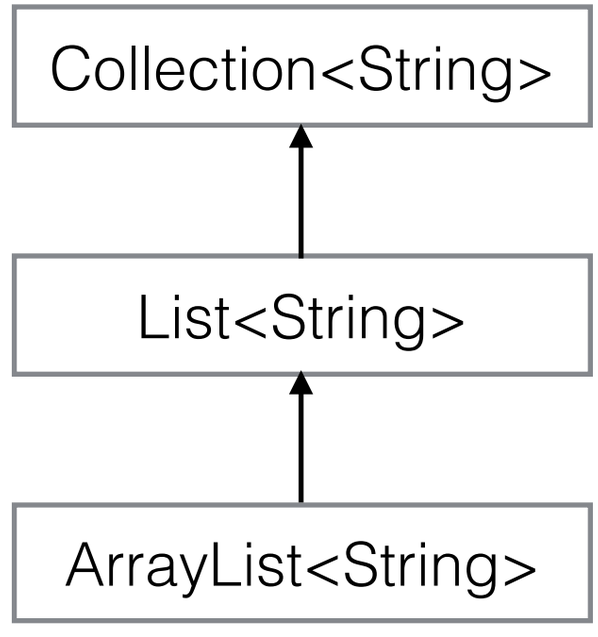
```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}
public class OrderedPair<K, V> implements Pair<K, V> {
    private K key;
    private V value;
    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

Generické typy a podtypy









protipříklad:

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(2);
List<Number> nums = ints;
nums.add(3.14);
Integer x=ints.get(1);
```

Modularita

- Umožňuje znovupoužívání kódu.
- Lze ji vnímat v různých rovinách abstrakce (metody, třídy, balíčky).
- Pokud je navržený program modulární, pak je obvykle dobře spravovatelný.

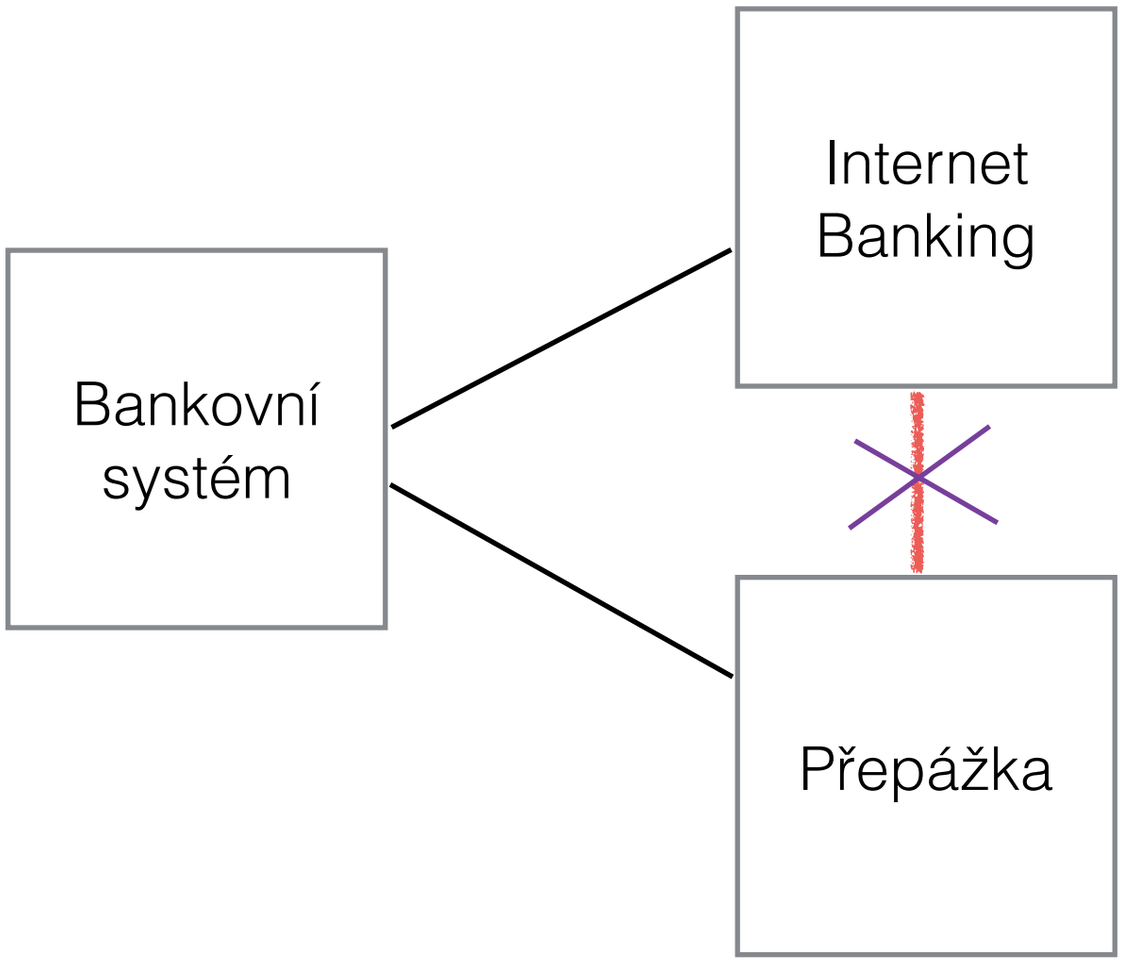
Modularita

- Umožňuje zaměnitelnost jednotlivých “modulů”.
- Výhodou je, že můžeme mít jednu architekturu, která pouhou výměnou jednotlivých modulů může plnit jinou funkci.
- Zpřehledňuje a snižuje množství kódu.
- Jednotlivé moduly by měly být na sobě co nejméně závislé (závislost pouze pomocí rozhraní).

Law of Demeter

- Pravidlo, které nám říká následující:
  - Každá jednotka by měla mít jen omezené znalosti o dalších jednotkách: pouze jednotky úzce spjaté s touto jednotkou.
  - Každá jednotka by měla mluvit pouze se svými přáteli a nemluvit s cizími jednotkami.
  - Mluvit jen s bezprostředními přáteli.
- Minimalizuje provázanost jednotlivých modulů.

Příklad: Modularita



Law of Demeter

- Z hlediska OOP můžeme toto pravidlo aplikovat pomocí následujících 5 pravidel pro posílání zpráv metodou M v objektu O
  - O sobě (this)
  - parametrům metody M
  - objektům, které byly v rámci metody M vytvořeny/instanciovány
  - komponentám, které jsou součástí O
  - globálním proměnným přístupným objektem O v rozsahu metody M

Zapouzdření

- Nám umožňuje řídit přístupová práva k metodám, třídám a objektům.
- Je založeno na předpokladu, že správné fungování tříd je založeno na řadě invariantů (vlastnostech, které platí po celou dobu existence objektu).
  - Programátor, který třídu používá nemůže tyto invarianty všechny znát.
  - Z tohoto důvodu je dobré programátorovi zamezit v přístupu k členským proměnným a metodám, které by mohly nesprávným použitím tyto invarianty porušit.

Zapouzdření

- Zapouzdřením se snažíme zabránit problému zpřístupnění implementace (Representation Exposure).

Příklad : CharSet - Specifikace

// Overview: A CharSet is a finite mutable set of Characters

// effects: creates a fresh, empty CharSet
public CharSet ( )

// modifies: this
// effects: $this_{post} = this_{pre} \cup \{c\}$
public void insert (Character c);

// modifies: this
// effects: $this_{post} = this_{pre} - \{c\}$
public void delete (Character c);

// returns: ($c \in this$)
public boolean member (Character c);

// returns: cardinality of this
public int size ( );

Implementace

```
class CharSet {
  private List<Character> elts
     = new ArrayList<Character>();
  public void insert(Character c)    {
     elts.add(c);
  }
  public void delete(Character c)    {
     elts.remove(c);
  }
  public boolean member(Cha
     return elts.contains(c)
  }
  public int size() {
     return elts.size();
  }
}
```

```
CharSet s = new CharSet();
Character a
     = new Character('a');
s.insert(a);
s.insert(a);
s.delete(a);
if (s.member(a))
     // print "wrong";
else
     // print "right";
```

Kde je chyba?

- Může být špatně metoda **delete**
  - Měla by odstranit všechny výskyty znaku.
- Může být špatně metoda **insert**
  - Neměla by vložit znak pokud již v Listu je.
- Měli bychom se to dozvědět z **invariantu reprezentace** datového typu.

Representation Invariant

- Specifikuje stav ve kterém platí, že je datová struktura dobře utvořená.

- Zachycuje informaci, která musí platit napříč všemi metodami.

- Můžeme ho specifikovat např. takto:

```
class CharSet {
  // Rep invariant: elts has no nulls and no duplicates
  private List<Character> elts;
  ...
```

Nebo formálněji:

$\forall$ indices i of elts . elts.elementAt(i) $\neq$ null
$\forall$ indices i, j of elts .
    i $\neq$ j $\Rightarrow \neg$ elts.elementAt(i).equals(elts.elementAt(j))

Nyní můžeme nalézt chybu

```
// Rep invariant:
// elts has no nulls and no duplicates

public void insert(Character c) {
   elts.add(c);
}

public void delete(Character c) {
   elts.remove(c);
}
```

Další ukázka invariantu

```
class Account {
    private int balance;
    // history of all transactions
    private List<Transaction> transactions;
    …
}
```

// real-world constraints:
balance ≥ 0
balance = Σ_i transactions.get(i).amount
// implementation-related constraints:
transactions ≠ null
no nulls in transactions

CharSet pokračování

- Uvažujme přidání následující metody do třídy CharSet

```
// returns: a List containing the members of this
public List<Character> getElts();
```

- S následující implementací

```
// Rep invariant: elts has no nulls and no duplicates
public List<Character> getElts()  { return elts; }
```

- Dodržuje implementace getElts() rep invariant?

Representation Exposure

- Uvažujme následující kód

```
CharSet s = new CharSet();
Character a = new Character('a');
s.insert(a);
s.getElts().add(a);
s.delete(a);
if (s.member(a)) …
```

- Zpřístupnění implementace je tedy externí přístup k členské proměnné.

- Zpřístupnění implementace je téměř vždy problém (může dojít k porušení invariantu)! Snažíme se mu vyhnout.

Způsoby jak zabránit Representation Exposure

1. Využít immutabilitu

```
Character choose() {
  return elts.elementAt(0);
}
```

Character is immutable.

2. Vytvořit kopii

```
List<Character> getElts() {
  return new ArrayList<Character>(elts);
  // or: return (ArrayList<Character>) elts.clone();
}
```

Mutating a copy doesn't affect the original.

3. Vytvořit immutable kopii

```
List<Character> getElts() {
  return Collections.unmodifiableList<Character>(elts);
}
```

Client cannot mutate

Literatura

- http://fi.muny.cz/data/PB006/PB006_slides2009.pdf

- https://edux.feld.cvut.cz/courses/X36OBP/_media/lectures/02-subtyping.pdf

Literatura

- http://courses.cs.washington.edu/courses/cse331/11wi/lectures/lect05-af-ri.pdf