

OMO

3 - Klíčové koncepty modelování systémů II

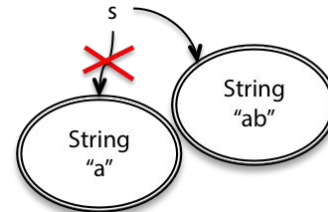
- Abstraktní datový typ
- Mutabilita, immutabilita
- Invarianty

Mutabilita

Mutable (měnitelné) objekty jsou takové, které obsahují metody, které modifikují hodnotu objektu.

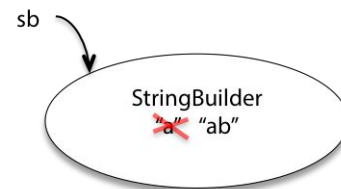
String je immutable, tedy např. pro přidání znaků na konec se vždy vytváří nový String.

```
String s = "a";  
s = s.concat("b"); // s+="b" and s=s+"b" also mean the same thing
```



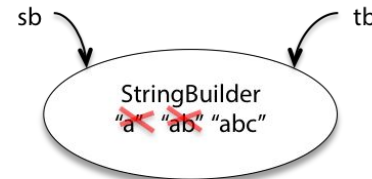
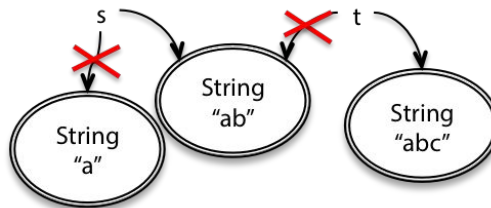
StringBuilder je mutable. Tedy i přidání znaků na konec modifikuje originální objekt.

```
StringBuilder sb = new StringBuilder("a");  
sb.append("b");
```



Rozdíl mezi mutable a immutable typy začne být evidentní ve chvíli, kdy máme víc jak jednu referenci na objekt. Např. když `t` ukazuje na stejný String jako `s`, a `tb` ukazuje na StringBuilder jako `sb`, tak:

```
String t = s;  
t = t + "c";  
StringBuilder tb = sb;  
tb.append("c");
```



Rizika mutability

Mutable typy jsou “mocnější” z hlediska funkcionality a zpravidla mají i lepší performance - immutable typy při většině volání vytvářejí kopie.

Proč tedy vůbec používat immutable typy?

- **immutable typy jsou méně náchylné pro vznik bugů** - u mutable typů vzniká velké množství chyb kvůli tzv. Aliasingu - na ten samý objekt se odkazuje z různých míst kódu.-
- **kód s immutable typy je jednodušší na pochopení** - čtenář kódu nemusí studovat implementaci, aby zjistil co se vlastně děje na pozadí
- **kód s immutable typy i vlastní immutable typ je jednodušší na upravování** - jestliže mám pod kontrolou co se děje na pozadí v kódu, tak je mnohem bezpečnější zasahovat do již hotového kódu
- **Mutable objekty zesložitují kontrakt a zhoršují reuse** - při přepoužití musím zpravidla zasahovat mnohem hlouběji do kódu a řešit situace Aliasingu.

I když obecné doporučení je používat immutable typy, tak jsou určité situace, kdy je výhodnější použít mutable typ (např. z performance důvodů) a existují užitečné mutable typy jako např: z Java Collections API jako je ArrayList, Hashtable.

Collections API poskytuje metody, které vrací immutable varianty: např. `Collections.unmodifiableList`

Rizika mutability - předávání mutable parametrů

Vytvoříme metodu, která sečte hodnoty v listu

```
/** @return the sum of the numbers in the list */  
public static int sum(List<Integer> list) {  
    int sum = 0;  
    for (int x : list)  
        sum += x;  
    return sum;  
}
```

Nakonec obě metody použijeme

```
// meanwhile, somewhere else in the code...  
public static void main(String[] args) {  
    // ...  
    List<Integer> myData = Arrays.asList(-5, -3, -2);  
    System.out.println(sumAbsolute(myData));  
    System.out.println(sum(myData));  
}
```

Pak vytvoříme metodu pro sečtení absolutních hodnot v seznamu (nebo jen v dobré víře optimalizujeme existující metodu). Abychom dodrželi principy DRY a nemuseli draze vytvářet list znovu, tak modifikujeme původní list

```
/** @return the sum of the absolute values of the list */  
public static int sumAbsolute(List<Integer> list) {  
    // Let's reuse sum(), because of DRY  
    for (int i = 0; i < list.size(); ++i)  
        list.set(i, Math.abs(list.get(i)));  
    return sum(list);  
}
```

=>

- Náchylné na chyby: Těžko vinit implementátora `sumAbsolute()`, chtěl jen dodržet princip DRY a nechtěl vytvářet drahou kopii celého seznamu. Je jen otázkou času než si některý programátor neuvědomí, že tato metoda modifikuje původní list.
- Těžko srozumitelné: Když budete procházet kód v `main()`, těžko vás napadne, že se seznam změní mezi voláním `sumAbsolute(myData)` a `sum(myData)`

Rizika mutability - vracení mutable parametrů

Tato metod vrací první jarní den (volá externí službu) na objektu Java *Date* (je mutable)

```
public static Date startOfSpring() {  
    return askExternalService();  
}
```

Vývojáři začnou tento kód používat pro naplánování party

```
public static void partyPlanning() {  
    Date partyDate = startOfSpring();  
    // ...  
}
```

Pak se první vývojář rozhodne, že bude odpověď ze služby cachovat, aby se nepřetěžovala

```
public static Date startOfSpring() {  
    if (groundhogAnswer == null) groundhogAnswer = askGroundhog();  
    return groundhogAnswer;  
}  
  
private static Date groundhogAnswer = null;
```

A druhý vývojář se rozhodne, že posune party o jeden měsíc

```
public static void partyPlanning() {  
    // Let's have a party one month after spring starts!  
    Date partyDate = startOfSpring();  
    partyDate.setMonth(partyDate.getMonth() + 1);  
    // => Next call returns different date (+1)  
}
```

=>

- Náchylné na chyby: Opět je vytvořen kód, který jen čeká na zavlečení chyby
- Náročné na změny: Dva vývojáři provedli dvě úpravy což vyústilo v chybu

Abstrakce - datová abstrakce

Odděluje abstraktní vlastnosti datového typu od jeho implementace. Abstraktní vlastnosti jsou ty, které jsou viditelné a měl bych je brát v potaz v kódu, ve kterém abstraktní datový typ používám. Konkrétní implementace je schovaná a mohu ji bez dopadu na tento kód měnit.

Hlavním reprezentantem datové abstrakce je **Abstraktní Datový Typ (ADT)**. ADT je matematický model pro datový typ definovaný množinou hodnot a operací nad těmito hodnotami, které splňují definované axiomy. Ekvivalentní k algebraické struktuře v abstraktní algebře.

Příklad: Typ Integer je ADT definovaný hodnotami ..., -2, -1, 0, 1, 2, ... a operacemi +, -, /, <, >, = které splňují axiomy asociativity, komutativity atd. V programovacích jazycích např. typ *boolean*, *float* atd. reprezentuje tzv. ADT.

Další příklady:

- V objektově orientovaném programování jsou třídy reprezentantem datové abstrakce - zapouzdřují přístup k datům (encapsulation) a poskytují pouze část dat (information hiding)
- Typy z collection API jako *Collection*, *List*, *Set*, *Map* jsou příkladem datové abstrakce - předepisují práci s datovou strukturou pomocí API a definuje princip práce s daty.
- RDBMS používají abstrakci tabulky, které má záznamy (*Row*) a sloupcečky (*Column*). Uživatel je odstíněn

Mutable a immutable ADT

Typy, vestavěné nebo uživatelem definované, lze klasifikovat jako:

- **mutable** (měnitelné)
- **immutable** (neměnné).

Mutable typy lze změnit, to znamená, že poskytují operace, které při spuštění způsobují, že výsledky dalších operací na stejném objektu dávají různé výsledky.

=> `Date` je mutable, protože můžete volat `setMonth` a sledovat změnu pomocí operace `getMonth`.

=> `String` je neměnný, protože jeho operace vytvářejí nové objekty `String` spíše než by měnily existující.

Pozn. Někdy je typ poskytován ve dvou podobách, v měnitelném a nezměnitelné formě. Například `StringBuilder` je mutable verze `String` (ačkoli oba nejsou určitě stejným Java typem a nejsou vzájemně zaměnitelné).

Clasifikace typů a operací

Operace abstraktního typu dělíme takto:

- **Creators** - vytvářejí nové objekty daného typu. Creator může vzít objekt jako argument, nikoli však objekt stejného typu jako ten, kterého vytváří.
- **Producers** - vytvářejí nové objekty ze starých objektů daného typu. Operace `concat` v třídě `String` je příklad produceru: vezme dva stringy a vytvoří z nich nový reprezentující jejich spojení
- **Observers** - berou objekty abstraktního typu a vracejí objekty jiného typu. Operace `size` v třídě `List` je příkladem observeru, jelikož vrací `int`.
- **Mutators** - mění objekty. Metoda `add` třídy `List`, je příkladem mutátoru, protože mění list tím, že na jeho konec přidává element.

Schématicky lze tyto různé typy zapsat následovně

- creator : $t^* \rightarrow T$
- producer : $T+, t^* \rightarrow T$
- observer : $T+, t^* \rightarrow t$
- mutator : $T+, t^* \rightarrow \text{void} \mid t \mid T$

Kde T je sám abstraktní typ; každé t je nějaký jiný typ. $+$ značí, že se typ může v signatuře metody vyskytnout jednou nebo n-krát, $*$ a je nula nebo n-krát, $|$ značí *nebo*.

Příklady pro pochopená schéma

Producer vezme dvě hodnoty abstraktního typu T - [String.concat\(\)](#):

- [concat](#) : $\text{String} \times \text{String} \rightarrow \text{String}$

Observer může být bezparametrický (žádný argument typu t):

- [size](#): $\text{List} \rightarrow \text{int}$

... nebo naopak brát několik parametrů:

- [regionMatches](#) : $\text{String} \times \text{boolean} \times \text{int} \times \text{String} \times \text{int} \times \text{int} \rightarrow \text{boolean}$

Creator operace je často implementována jako *constructor* , např [new ArrayList\(\)](#),

- [new ArrayList\(\)](#): $() \rightarrow \text{ArrayList}$

může to ale být i jednoduchá statická metoda jako [Arrays.asList\(\)](#):

- [valueOf\(\)](#): $() \rightarrow \text{String}$ Pozn. Creator implementovaný jako statická metoda je často nazýván jako *factory method* . Např. [String.valueOf](#).

Mutators často vrací void, ale není to ve 100%. Metoda, která vrací void musí být volána kvůli nějakému side-efektu. Např. [Component.add\(\)](#) v Java UI toolkitu vrací samotný objekt, aby bylo možné metody `add()` zřetěžit za sebe.

- [add](#): $() \rightarrow \text{Component}$

Příklady z Javy

`int` je primitivní datový typ. `int` je immutable, nemá tedy žádné mutators.

- creators: čísla 0, 1, 2, ...
- producers: arithmetické operátory +, -, *, /
- observers: porovnávací operátory ==, !=, <, >
- mutators: nemá

`List` je typ pro reprezentaci listu. `List` je mutable. `List` je také interface, což znamená, že ho implementují ostatní třídy, např. `ArrayList` a `LinkedList`.

- creators: `ArrayList` a `LinkedList` konstruktory, [Collections.singletonList](#)
- producers: [Collections.unmodifiableList](#)
- observers: `size`, `get`
- mutators: `add`, `remove`, `addAll`, [Collections.sort](#)

`String` je typ pro reprezentaci řetězce. `String` je immutable.

- creators: `String` konstruktory
- producers: `concat`, `substring`, `toUpperCase`
- observers: `length`, `charAt`
- mutators: nemá

Jak navrhovat ADT

- Je lepší mít **minimální množinu jednoduchých operací, které lze dobře kombinovat**, než spoustu složitých operací.
- Každá operace by měla mít **přesně vymezený účel** fungující bez výjimek ve 100 % use casů. Např. operace `sum` nemá být definována v třídě `List`, protože ačkoliv funguje pro seznam integers, tak nedává smysl pro stringy. Úplně stejně `sum` nedává smysl pro vnořený list.
- Množina operací by měla **dobře pokrývat výpočty, které s typem budou klienti provádět**.
- Dobrým **testem pro ověření, že typ poskytuje dostatečné metody, je zkusit postupně získat všechny vlastnosti typu**. Pokud je to jednoduché, tak je typ navržen dobře. Např. metoda `size` není nezbytně nutná - velikost mohu zjistit i iterováním přes celý list, nicméně je to extrémně neefektivní
- Dobrým **testem pro ověření, že typ neposkytuje nadbytek operací, je zkusit postupně metody odebírat a zkoušet jestli mohu rozumně získat vlastnosti objektu**
- Typ by měl být buď generický: seznam, množina, graf atd. nebo doménově specifický: mapa ulic, databáze zákazníků, telefonní seznam. **Meměl by kombinovat generické a doménově specifické vlastnosti**. Např. typ `Balíček karet` by neměl mít metodu `add`, která umožňuje přidávat libovolné typy jiné než karty. Naopak generický list by neměl mít metodu `dealCards`.

ADT a nezávislost na reprezentaci

ADT by měl být nezávislý na reprezentaci - tedy na aktuální datové struktuře a datových proměnných - tak, aby změny v reprezentaci neměly vliv na kód mimo typ. Např. operace poskytované třídou `List` jsou nezávislé na tom, jestli jsou data uvnitř reprezentována jakou prolínkovací seznam `LinkedList` nebo pole

```
/** MyString represents an immutable sequence of characters. */
public class MyString {
    //////////////// Example of a creator operation ////////////////
    /** @param b a boolean value
     * @return string representation of b, either "true" or "false" */
    public static MyString valueOf(boolean b) { ... }
    //////////////// Examples of observer operations ////////////////
    /** @return number of characters in this string */
    public int length() { return 0 }
    /** @param i character position (requires 0 <= i < string length)
     * @return character at position i */
    public char charAt(int i) { ... }
    //////////////// Example of a producer operation ////////////////
    /** Get the substring between start (inclusive) and end (exclusive).
     * @param start starting index
     * @param end ending index. Requires 0 <= start <= end <= string length.
     * @return string consisting of charAt(start)..charAt(end-1) */
    public MyString substring(int start, int end) { ... }
}
```

ADT a nezávislost na reprezentaci

Pro náš nový ADT napíšeme implementaci. Náš ADT začnou používat vývojáři. Poté provedeme performance optimalizaci kódu. U správně implementovaného ADT tato optimalizace nebude vyžadovat zásah uživatelů do jejich kódu.

```
private char[] a;
...
public static MyString valueOf(boolean b) {
    MyString s = new MyString();
    s.a = b ? new char[] { 't', 'r', 'u', 'e' }
           : new char[] { 'f', 'a', 'l', 's', 'e' };
    return s;
}

public int length() {return a.length;}

public char charAt(int i) { return a[i];}

public MyString substring(int start, int end) {
    MyString that = new MyString();
    that.a = new char[end - start];
    System.arraycopy(this.a, start, that.a, 0, end -
start);
    return that;
}
```

=>

```
private char[] a;
private int start;
private int end;
...
public static MyString valueOf(boolean b) {
    MyString s = new MyString();
    s.a = b ? new char[] { 't', 'r', 'u', 'e' }
           : new char[] { 'f', 'a', 'l', 's', 'e' };
    s.start = 0;
    s.end = s.a.length;
    return s;
}

public int length() { return end - start;}
public char charAt(int i) { return a[start + i];}
public MyString substring(int start, int end) {
    MyString that = new MyString();
    that.a = this.a;
    that.start = this.start + start;
    that.end = this.start + end;
    return that;
}
```

ADT a nezávislost na reprezentaci

Pro náš nový ADT napíšeme implementaci. Náš ADT začnou používat vývojáři. Poté provedeme performance optimalizaci kódu. U správně implementovaného ADT tato optimalizace nebude vyžadovat zásah uživatelů do jejich kódu což je síla nezávislosti na reprezentaci.

```
private char[] a;
...
public static MyString valueOf(boolean b) {
    MyString s = new MyString();
    s.a = b ? new char[] { 't', 'r', 'u', 'e' }
           : new char[] { 'f', 'a', 'l', 's', 'e' };
    return s;
}

public int length() {return a.length;}

public char charAt(int i) { return a[i];}

public MyString substring(int start, int end) {
    MyString that = new MyString();
    that.a = new char[end - start];
    System.arraycopy(this.a, start, that.a, 0, end -
start);
    return that;
}
```

=>

```
private char[] a;
private int start;
private int end;
...
public static MyString valueOf(boolean b) {
    MyString s = new MyString();
    s.a = b ? new char[] { 't', 'r', 'u', 'e' }
           : new char[] { 'f', 'a', 'l', 's', 'e' };
    s.start = 0;
    s.end = s.a.length;
    return s;
}

public int length() { return end - start;}
public char charAt(int i) { return a[start + i];}
public MyString substring(int start, int end) {
    MyString that = new MyString();
    that.a = this.a;
    that.start = this.start + start;
    that.end = this.start + end;
    return that;
}
```

ADT - Invarianty

Invariance ([česky](#) neměnnost) je označení pro situaci, v níž jsou jisté objekty neměnné při určitých událostech. Příkladem invariance je situace, kdy je dán systém veličin, které na sobě nějakým způsobem závisí. Potom se jedna z těchto veličin nazývá invariantní vůči změně jiné (referenční) veličiny, pokud má stále stejnou hodnotu při jakýchkoli změnách referenční veličiny.

Správný ADT zachovává své invarianty - *invarianta je vlastnost programu, která je vždy true pro jakýkoliv runtime stav programu a to nezávisle na chování klienta. Příkladem důležité invarianty je imutabilita: jakmile je immutable objekt vytvořen, tak si drží tu samou hodnotu.* Je pak výrazně jednodušší pracovat s kódem jestliže máme garantovanou tuto vlastnost.

```
public class Tweet {
    public String author;
    public String text;
    public Date timestamp;
    /**
     * Make a Tweet.
     */
    public Tweet(String author, String text, Date timestamp) {
        this.author = author;
        this.text = text;
        this.timestamp = timestamp;
    }
}
```

Jak zajistíme, že jednotlivé tweety jsou immutable?

ADT - immutable invarianty

Prvním problémem je, že objekty mohou přímo modifikovat proměnné třídy, jedná se o tzv. **Representation exposure** (vystavení reprezentace). To kromě toho, že ohrožuje invarianty, tak ohrožuje i **representation independence** (nezávislost na reprezentaci). Prvním krokem je tedy schování proměnných za *getter/setter* operace.

```
public class Tweet {  
  
    private final String author;  
    private final String text;  
    private final Date timestamp;  
  
    public Tweet(String author, String text, Date timestamp) {  
        this.author = author;  
        this.text = text;  
        this.timestamp = timestamp;  
    }  
    public String getAuthor() {  
        return author;  
    }  
    public String getText() {  
        return text;  
    }  
    public Date getTimestamp() {  
        return timestamp;  
    }  
}
```

Modifikátor *final* zajistí, že ani implementátor třídy nemůže upravovat stav objektu.

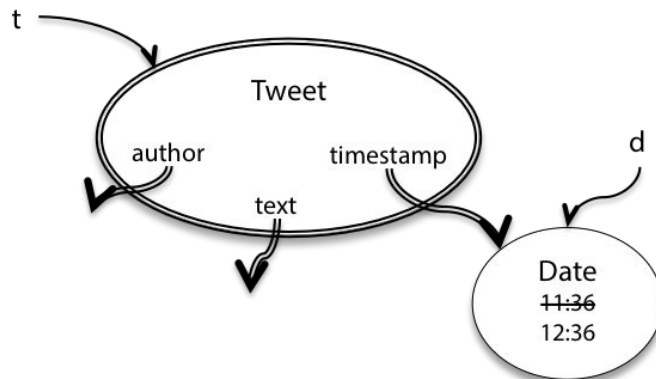
ADT - immutable invarianty

Tím to ale nekončí. Vývojář napíše následující kód:

```
/** @return a tweet that retweets t, one hour later*/  
public static Tweet retweetLater(Tweet t) {  
    Date d = t.getTimestamp();  
    d.setHours(d.getHours()+1);  
    return new Tweet("rbm11r", t.getText(), d);  
}
```

Čímž jsme z objektu tweet vrátili mutable objekt, který je pak modifikován ve vnějším kódu. To můžeme vyřešit jednoduchým patchem, tzv. *defensive copying*.

```
public Date getTimestamp() {  
    return new Date(timestamp.getTime());  
}
```



ADT - immutable invarianty

Vývojář ale může napsat i takovýto kód:

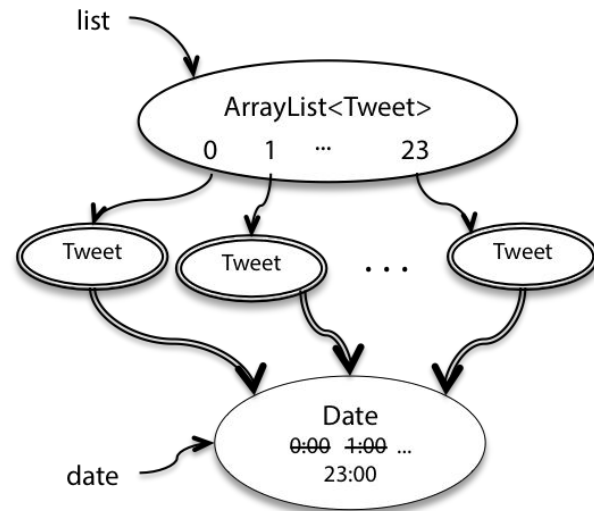
```
/** @return a List of 24 inspiring tweets, one per hour today*/  
public static List<Tweet> tweetEveryHourToday () {  
    List<Tweet> list = new ArrayList<Tweet>();  
    Date date = new Date();  
    for (int i = 0; i < 24; i++) {  
        date.setHours(i);  
        list.add(new Tweet("rbmlr", "keep it up! you can do it", date));  
    }  
    return list;  
}
```

Čímž nám všechny instance tweetu odkazují na ten samý date objekt.

```
public Tweet(String author, String text, Date timestamp) {  
    this.author = author;  
    this.text = text;  
    this.timestamp = new Date(timestamp.getTime());  
}
```

Pokud selže úplně vše, tak nebo nechceme dělat kopie objektů, tak je nutné to zapsat alespoň ve specifikaci

```
/**Make a Tweet.  
 * @param author Twitter user who wrote the tweet  
 * @param text text of the tweet  
 * @param timestamp date/time when the tweet was sent. Caller must never mutate this Date  
 object again! */  
public Tweet(String author, String text, Date timestamp) {
```



Problém fixujeme kopíí objektu Date v konstruktoru.

ADT - rep invarianty

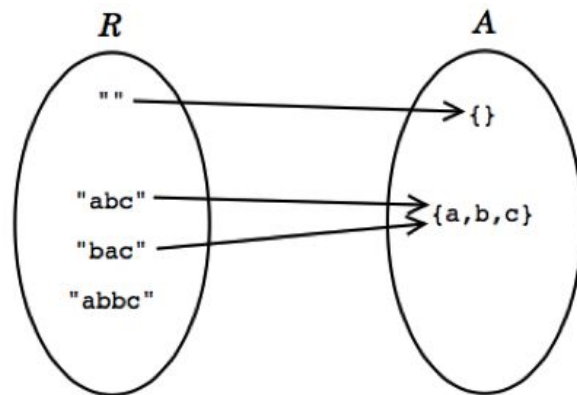
Na ADT je možné se dívat jako na vztah dvou prostorů hodnot. Prostor abstraktních hodnot a prostor reprezentací. Abstraktní prostor využíváme, protože to je právě to, co by měl ADT implementovat a vystavit

Implementujeme ADT CharSet. Pro vnitřní reprezentaci použijeme typ String.

```
public class CharSet {  
    private String s;  
    ...  
}
```

Pak rep prostor R obsahuje *Stringy* a abstraktní prostor A je matematická reprezentace množiny znaků

- Každá abstraktní hodnota je mapovaná na nějakou hodnotu z Rep prostoru. Ve finále musíme být schopni vytvářet a manipulovat s veškerými možnými hodnotami abstraktního prostoru a zároveň být schopni je reprezentovat.
- Některé abstraktní hodnoty jsou mapované na více reprezentačních hodnot. Je více možných způsobů jak reprezentovat neseříděnou množinu znaků jako string.
- Ne všechny rep hodnoty jsou mapované. Pokud string nemá duplicitu, tak to např. umožňuje ukončit *remove* metodu po nalezení první instance, protože další neexistuje.



ADT - rep invarianty

TO BE CONTINUED...