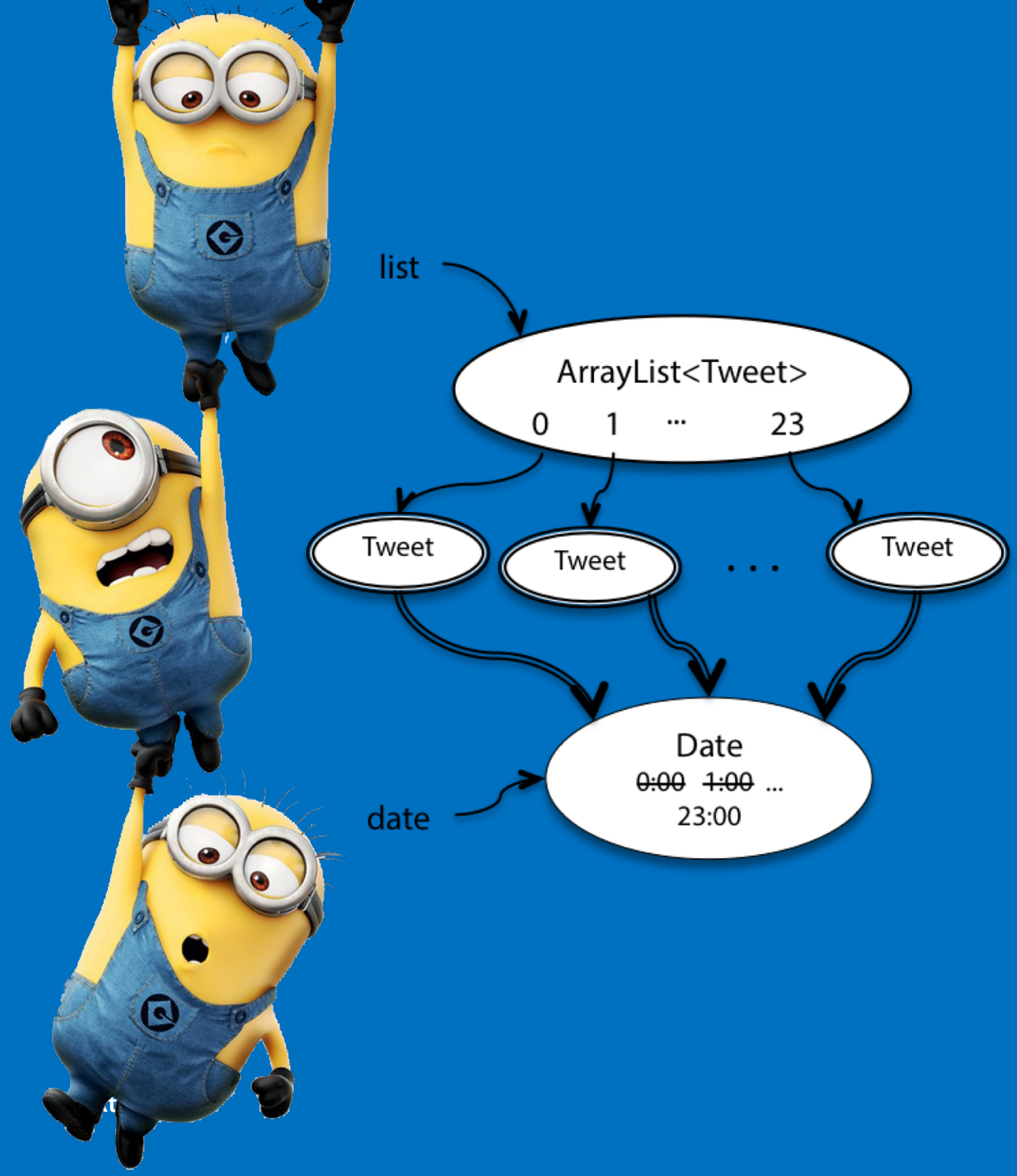


Search . . .

04

Klíčové koncepty modelování II

- Mutabilita, immutabilita
- Abstraktní Datové Typy
- Invarianty

kadlecd@fel.cvut.cz, www.czm.fel.cvut.cz

04 ADT – Abstraktní Datový Typ

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.

Příklady:

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.

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 používájí abstrakci tabulky, které má záznamy (*Row*) a sloupečky (*Column*). Uživatel je odstíněn.

04 ADT – Typy a Operace

Operace abstraktního datového 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.

04 ADT – Typy a Operace

Schématicky lze tyto různé typy zapsat následovně:

- **creator : t* → T**
- **producer : T+, t* → T**
- **observer : T+, t* → t**
- **mutator : T+, t* → void | t | 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*.

04 Příklady pro pochopení schéma

- Producer vezme dvě hodnoty abstraktního typu *T* - String.concat():

  concat : String × String → String

- Observer může být bezparametrický (žádný argument typu *t*):

  size: List → int

- … nebo naopak brát několik parametrů:

  regionMatches : String × boolean × int × String × int × int → boolean

- Creator operace je často implementována jako *constructor* , např new ArrayList(),

  new ArrayList(): () → ArrayList

- může to ale být i jednoduchá statická matoda jako Arrays.asList():

  valueOf: () → String

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() zřetězit za sebe.

04 Příklady pro pochopení schéma

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

04 NÁVRH 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 % případů**. NE v 99%. 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.
- Dobrým **testem pro ověřením, 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. **Neměl 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`.

04 INVARIANTY

Invarianta je vlastnost, která je splněna pro jakýkoliv runtime stav programu ve všech jeho stabilních stavech a nezávisí na chování klienta.

Pozn. Invariance by měla být zaručena pro volání public metod. Tedy mohou existovat mezistavy, kdy je invariance porušena.

Příklady:

- Immutabilita - jakmile je immutable objekt vytvořen, tak si drží tu samou hodnotu. Je pak výrazně jednodušší pracovat s kódem jestliže máme garantovanou tuto vlastnost.
- Binary search tree může mít invariantu, že pro všechny nody platí, že klíč levého potomka je menší než klíč samotného nodu. Správně implementovaná insert funkce tuto invariantu drží.
- Java *List* za každé situace drží pořadí prvků

04 INVARIANTY - Kontrola

Řešíme:

- zadokumentováním
- kontrolou v rámci volané metody
- speciální metodou, kterou voláme explicitně ve chvíli, kdy potřebujeme provést kontrolu *checkRep()*

```
class Account {
    // Rep invariant is
    // name != null && balance ≥ 0
   public void tranferMoney(int dstAccount, float amount) {
   ...
   }
```

```
class Account {
   public void tranferMoney(int dstAccount, float amount) {
     // using Java assert syntax
     assert balance ≥ 0 : "balance should be ≥ 0";
     // using junit.framework.Assert (you don't have to be writing a unit test to use this class)
     Assert.assertTrue ("balance should be ≥ 0", balance ≥ 0);
    }
}
```

```
class Child {
    private void checkRep( ) {
    assert (0 <= age < 18);
assert (birthdayge + age == todays_date);
assert (isLegalSSN(0 <= agsocial_security_number));
    }
}
```

04 ADT - immutable invarianty

Správný ADT zachovává své invarianty, dokonce je za to odpovědný. Důležitou invariantou vybraných ADT je immutabilita.

```
public class Tweet {
   public String author;
   public String text;
   public Date timestamp;

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

Jak zajistíme, že jednotlivé tweety jsou immutable?

04 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).

[1] Prvním krokem je tedy schování proměnných za *getter/setter* operace.

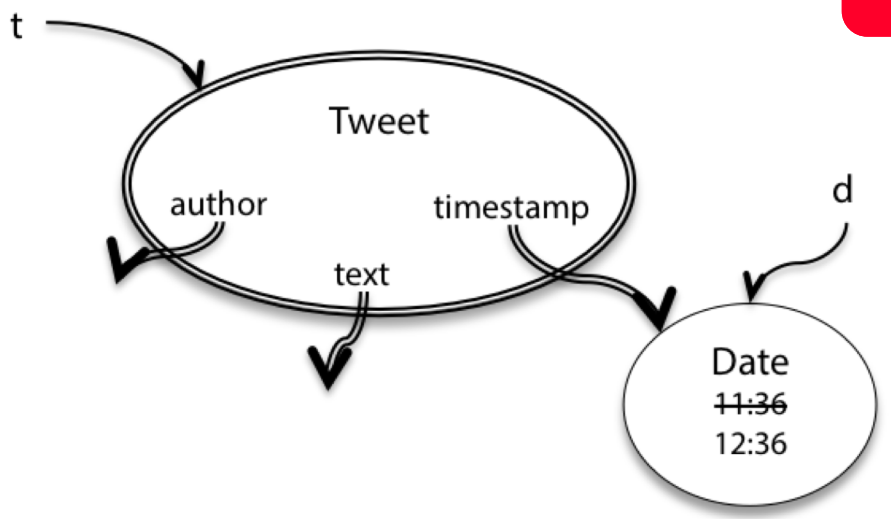
```
public class Tweet {
   private final String author;
   private final String text;
   private final Date timestamp;
   public Tweet(String author, String text, Date timestamp) {
       this.author = author;
       this.text = text;
       this.timestamp = timestamp;
   }
   public String getAuthor()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

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

04 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("rbmllr", t.getText(), d);
}
```



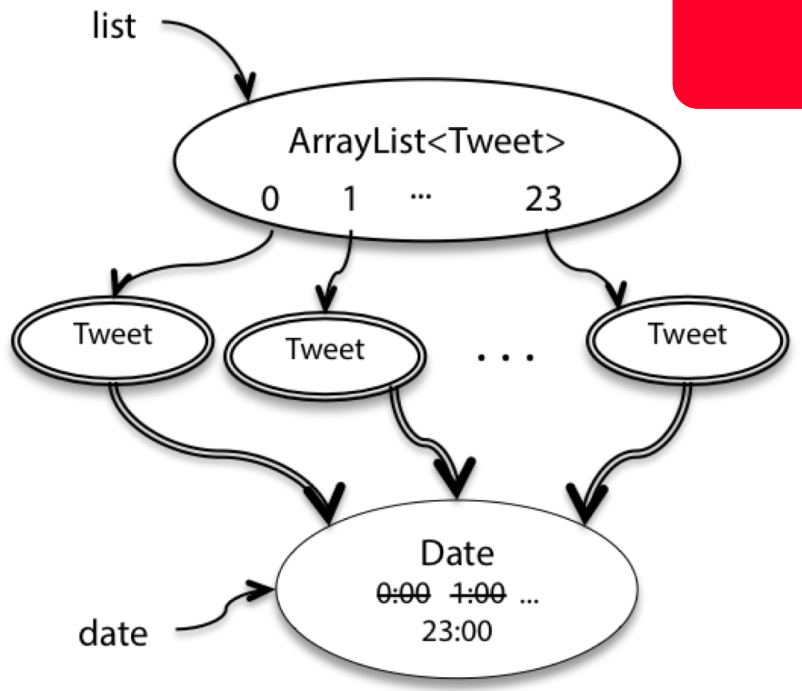
Čímž jsme z objektu tweet vrátili mutable objekt, který je pak modifikován ve vnějším kódu. To můžeme vyřešit jednoduchým patchem, tzv. ***defensive copying***.

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

04 ADT - immutable invarianty

Vývojář ale může napsat i takovýto kód:

```
/** @return a list of 24 inspiring tweets, one per hour*/
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("John", "You made 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());
}
```

Problém fixujeme kopií objektu Date v konstruktoru.

Pokud selže úplně vše 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. Caller must never mutate this Date object again! */
public Tweet(String author, String text, Date timestamp) {
```

04 ADT - immutable invarianty

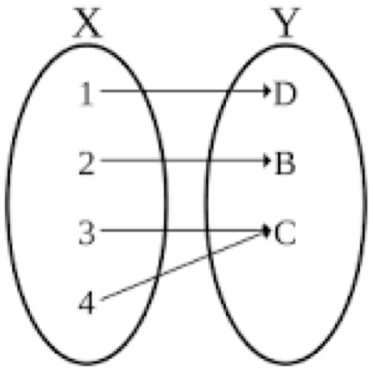
Na ADT je možné se dívat jako na vztah dvou prostorů hodnot. Prostor abstraktních hodnot **A** a prostor reprezentací **R** Abstraktní prostor využíváme, protože to je právě to, co by měl ADT implementovat a vystavit.

Definujeme dva základní termíny:

Abstraktní funkce - surjektivní mapování hodnoty z R do hodnot v A (abstraktních hodnot které reprezentují) - surjekce

AF : R → A

Pozn. Surjekce = zobrazení na, druh zobrazení mezi množinami, které zobrazuje na celou cílovou množinu. Každý prvek cílové množiny má tedy alespoň jeden vzor.



Rep invarianty - mapuje hodnoty z R na boolean (vlastnosti, které se drží a které ne)

RI : R → boolean

04 ADT - Rep(rezentační) invarianty

Příklad: Komplexní čísla - prostor reprezentací *R* je množina objektů třídy Complex a abstraktní prostor *A* je množina komplexních čísel.

```
class Complex {
    private double real;
    private double imag;
    // The abstraction function is
    //    real + i * imag
}
```

Příklad: Orientovaná úsečka

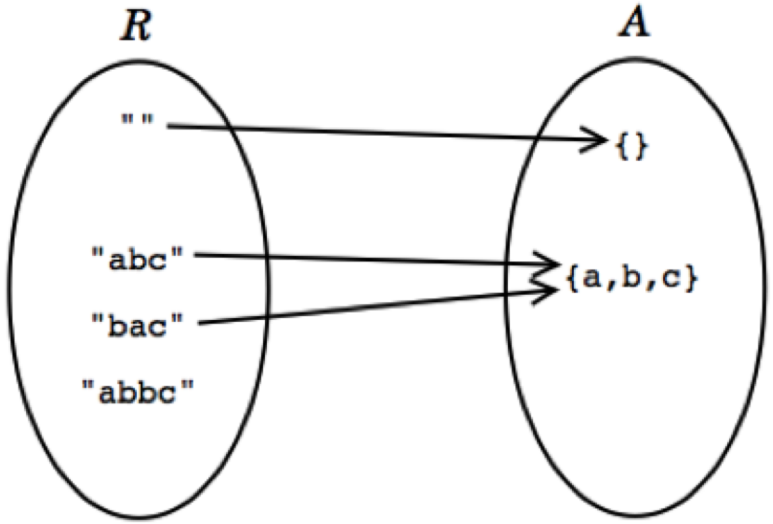
```
public class Line {
   private Point start;
   private double length;
   private double angle;

   // Abstraction function is
   //    AF(r) = line l such that
   //       l.start = r.start
   //       l.end.x = r.start.x + r.length * cos(r.angle)
   //       l.end.y = r.start.y + r.length * sin(r.angle)
   ...
}
```

04 ADT - immutable invarianty

Příklad: Implementujeme ADT CharSet. Pro vnitřní reprezentaci použijeme typ String.

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

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



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

04 ADT - immutable invarianty

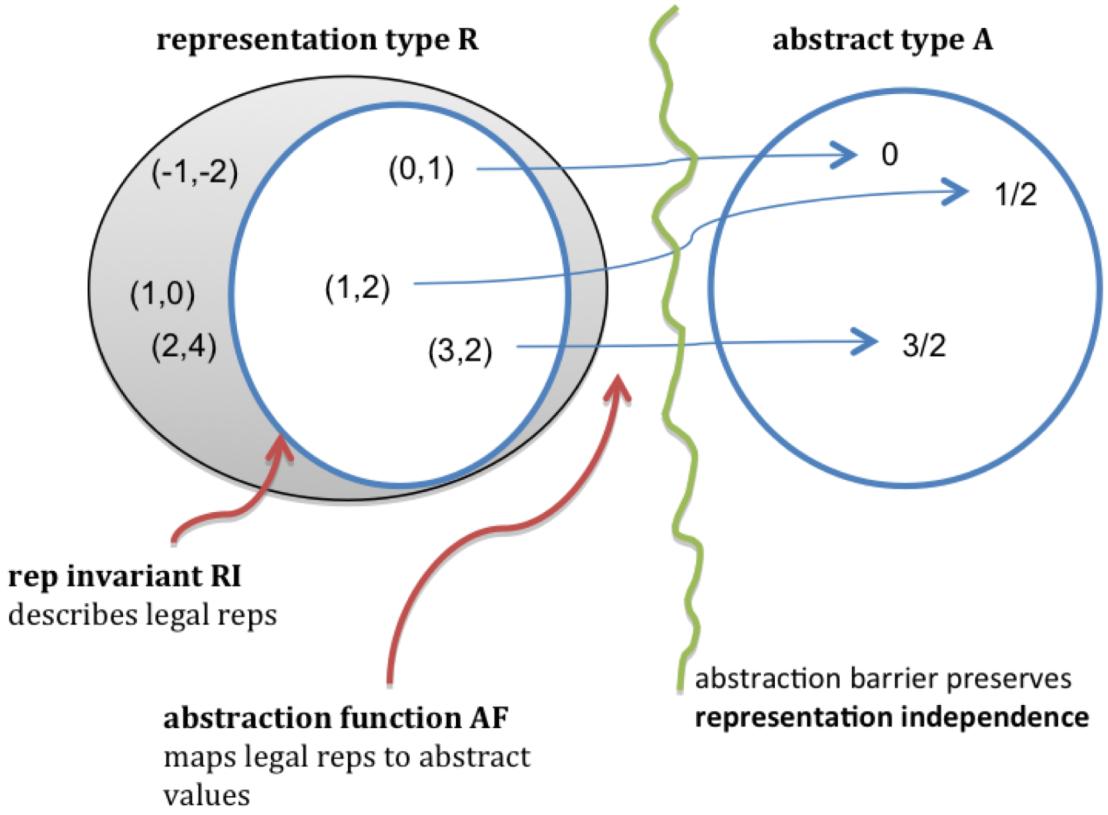
Příklad: Balíček karet

```java
/**
* Card represents an immutable playing card.
* suit : {Clubs,Diamonds,Hearts,Spades} // barvy
* value : {Ace,2,...,10,Jack,Queen,King} // typy karet
*/
public class Card {
   private int index;
    // Abstraction function is
   //     suit = S(index div 13) where S(0)=Clubs,S(1)=Diamonds,S(2)=Hearts,S(3)=Spades
   //     value = V(index mod 13) where V(1)=Ace, V(2)=2, ..., V(10)=10,V(11)=Jack,V(12)=Queen,V(0)=King
   // Rep invariant is
   //     0 <= index <= 51
   private void checkRep( ) {
      if (index < 0 || index > 51)
        throw new RuntimeException("card index out of range");
   }

   public Card(CardSuit barva, CardValue value) {
      ... // initialize Card
      checkRep( );
   }
 ...
}
```

04 ADT - immutable invarianty

Příklad ADT pro reprezentaci racionálních čísel =>

- Páry jako např. *(2,4)* a *(18,12)* nepatří do RI prostoru (mezi Rep Invarianty), protože nejsou v prvočíselném rozkladu jak je vyžadováno (**numer/denom is in reduced form)**




```java
public class RatNum {
    private final int numer;
    private final int denom;
    // Rep invariant:
    //   denom > 0
    //   numer/denom is in reduced form
    // Abstraction Function:
    //   represents the rational number numer / denom
    public RatNum(int n) {
      numer = n;
      denom = 1;
      checkRep();
    }
    public RatNum(int n, int d) throws ArithmeticException{
      // reduce ratio to lowest terms
      int g = gcd(n, d);
      n = n / g;
      d = d / g;
      // make denominator positive
      if (d < 0) {
        numer = -n;
        denom = -d;
      } else {
        numer = n;
        denom = d;
      }
      checkRep();
    }
}
```

04 ZÁVĚR

- Invarianta je vlastnost, která je vždy splněna na dané instanci ADT po celý jeho životní cyklus
- Dobrý ADT si drží své vlastní invarianty
- Creators a Producers nejdříve inicializují Invarianty a Observers a Mutators je poté drží.
- Rep invarianty specifikují validní hodnoty reprezentace a jsou kontrolovány v runtime speciální metodou `checkRep()`, která do jisté míry nahrazuje preconditions.
- Abstraktní funkce mapují konkrétní reprezentace na abstraktní hodnoty, které reprezentují
- Zpřístupnění reprezentace (***Representation exposure***) je ohrožením pro nezávislost reprezentace (***Representation independence***) a zachování invariant (***Invariant preservation)***.
- **ADT Rep Invarianty = kuchařka jak vytvářet abstraktní datové typy tak:**
    a. **aby se minimalizovaly možnosti vzniku chyb**
    b. **kód byl srozumitelný**
    c. **kód byl připravený na změny.**

REFERENCE: **https://ocw.mit.edu/ans7870/6/6.005/s16/classes/13-abstraction-functions-rep-invariants/#rep_invariant_and_abstraction_function**