

## 第三章第三节 抽象数据类型(ADT)

3-1节研究了“数据类型”及其特性；3-2节研究了方法和操作的“规约”及其特性；在本节中，我们将数据和操作复合起来，构成ADT，学习ADT的核心特征，以及如何设计“好的”ADT。

### Outline

- ADT及其四种类型
  - ADT的基本概念
  - ADT的四种类型
  - 设计一个好的ADT
- 表示独立性
- 不变量和表示泄露
- 抽象函数AF和表示不变量RI
  - AF与RI
  - 用注释写AF和RI

### Notes

#### ## ADT及其四种类型

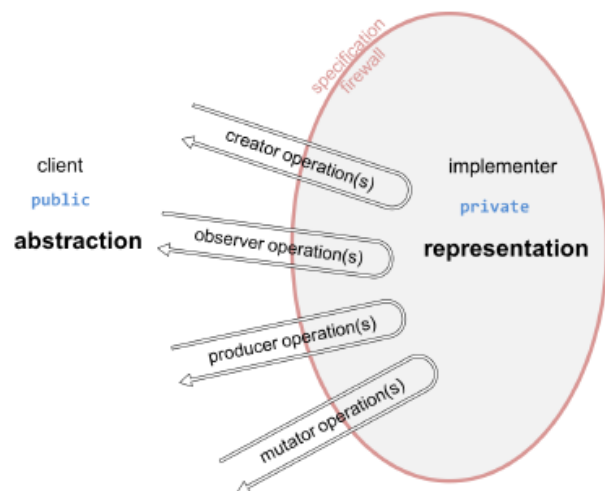
##### 【ADT的基本概念】

- 抽象数据类型（Abstract Data Type，ADT）是指一个数学模型以及定义在该模型上的一组操作；即包括数据元素，数据关系以及相关的操作。
- ADT具有以下几个能表达抽象思想的词：
  - 抽象化：用更简单、更高级的思想省略或隐藏低级细节。
  - 模块化：将系统划分为组件或模块，每个组件可以设计，实施，测试，推理和重用，与系统其余部分分开使用。
  - 封装：围绕模块构建墙，以便模块负责自身的内部行为，并且系统其他部分的错误不会损坏其完整性。
  - 信息隐藏：从系统其余部分隐藏模块实现的细节，以便稍后可以更改这些细节，而无需更改系统的其他部分。
  - 关注点分离：一个功能只是单个模块的责任，而不跨越多个模块。
- 与传统类型定义的差别：
  - 传统的类型定义：关注数据的具体表示。
  - 抽象类型：强调“作用于数据上的操作”，程序员和client无需关心数据如何具体存储的，只需设计/使用操作即可。
- ADT是由操作定义的，与其内部如何实现无关！

##### 【ADT的四种类型】

- 前置定义：mutable and immutable types
  - 可变类型的对象：提供了可改变其内部数据的值的操作。Date

- 不变数据类型： 其操作不改变内部值，而是构造新的对象。String



### Creators (构造器)：

- 创建某个类型的新对象，一个创建者可能会接受一个对象作为参数，但是这个对象的类型不能是它创建对象对应的类型。可能实现为构造函数或静态函数。（通常称为工厂方法）

- $t^* \rightarrow T$

- 栗子：Integer.valueOf( )

### Producers (生产器)：

- 通过接受同类型的对象创建新的对象。

- $T+, t^* \rightarrow T$

- 栗子：String.concat( )

### Observers (观察器)：

- 获取抽象类型的对象然后返回一个不同类型的对象/值。

- $T+, t^* \rightarrow t$

- 栗子：List.size( )；

### Mutators (变值器)：

- 改变对象属性的方法，

- 变值器通常返回void，若为void，则必然意味着它改变了对象的某些内部状态；当然，也可能返回非空类型

- $T+, t^* \rightarrow t \parallel T \parallel \text{void}$

- 栗子：List.add( )

- 解释：T是ADT本身；t是其他类型；+表示这个类型可能出现一次或多次；\*表示可能出现0次或多次。

- 更多栗子：

#### int is immutable, so it has no mutators.

- creators: the numeric literals 0 , 1 , 2 , ...
- producers: arithmetic operators + , - , \* , /
- observers: comparison operators == , != , < , >
- mutators: none (it's immutable)

#### String is Java's string type. String is immutable.

- creators: String constructors
- producers: concat , substring , toUpperCase
- observers: length , charAt
- mutators: none

| ADT concept        | Ways to do it in Java       | Examples  |
|--------------------|-----------------------------|---|
| Creator operation  | Constructor                 | <code>ArrayList()</code>  |
|                    | Static (factory) method     | <code>Collections.singletonList()</code> , <code>Arrays.asList()</code> |
|                    | Constant                    | <code>BigInteger.ZERO</code>  |
| Observer operation | Instance method             | <code>list.get()</code>   |
|                    | Static method               | <code>Collections.max()</code>  |
| Producer operation | Instance method             | <code>String.trim()</code>  |
|                    | Static method               | <code>Collections.unmodifiableList()</code>                             |
| Mutator operation  | Instance method             | <code>list.add()</code>   |
|                    | Static method               | <code>Collections.copy()</code>   |
| Representation     | <code>private</code> fields |   |

【设计一个好的ADT】

设计好的ADT，靠“经验法则”，提供一组操作，设计其行为规约 spec

- 原则 1：设计简洁、一致的操作。
  - 最好有一些简单的操作，它们可以以强大的方式组合，而不是很多复杂的操作。
  - 每个操作应该有明确的目的，并且应该有一致的行为而不是一连串的特殊情况。
- 原则 2：要足以支持用户对数据所做的所有操作需要，且用操作满足用户需要的难度要低。
  - 提供get()操作以获得list内部数据
  - 提供size()操作获取list的长度
- 原则 3：要么抽象、要么具体，不要混合 —— 要么针对抽象设计，要么针对具体应用的设计。

【测试ADT】


- 测试creators, producers, and mutators：调用observers来观察这些 operations的结果是否满足spec；
- 测试observers：调用creators, producers, and mutators等方法产生或改变对象，来看结果是否正确。

## 表示独立性

- 表示独立性：client使用ADT时无需考虑其内部如何实现，ADT内部表示的变化不应影响外部spec和客户端。
- 除非ADT的操作指明了具体的前置条件/后置条件，否则不能改变ADT的内部表示——spec规定了 client和implementer之间的契约。

【一个例子：字符串的不同表示】

让我们先来看看一个表示独立的例子，然后考虑为什么很有用，下面的MyString抽象类型是我们举出的例子。下面是规格说明：



```
1 /** MyString represents an immutable sequence of characters. */
2 public class MyString {
3
4     //////////////// Example of a creator operation ////////////////
5     /** @param b a boolean value
6      *   @return string representation of b, either "true" or "false" */
7     public static MyString valueOf(boolean b) { ... }
8
9     //////////////// Examples of observer operations ////////////////
10    /** @return number of characters in this string */
11    public int length() { ... }
12
13    /** @param i character position (requires 0 <= i < string length)
14     *   @return character at position i */
```

```

15     public char charAt(int i) { ... }
16
17     // Example of a producer operation
18     /** Get the substring between start (inclusive) and end (exclusive).
19      *   @param start starting index
20      *   @param end ending index. Requires 0 <= start <= end <= string length.
21      *   @return string consisting of charAt(start)...charAt(end-1) */
22     public MyString substring(int start, int end) { ... }
23 }

```



使用者只需要/只能知道类型的公共方法和规格说明。下面是如何声明内部表示的方法，作为类中的一个实例变量：

```
private char[] a;
```

使用这种表达方法，我们对操作的实现可能是这样的：



```

1 public static MyString valueOf(boolean b) {
2     MyString s = new MyString();
3     s.a = b ? new char[] { 't', 'r', 'u', 'e' }
4           : new char[] { 'f', 'a', 'l', 's', 'e' };
5     return s;
6 }
7
8 public int length() {
9     return a.length;
10 }
11
12 public char charAt(int i) {
13     return a[i];
14 }
15
16 public MyString substring(int start, int end) {
17     MyString that = new MyString();
18     that.a = new char[end - start];
19     System.arraycopy(this.a, start, that.a, 0, end - start);
20     return that;
21 }

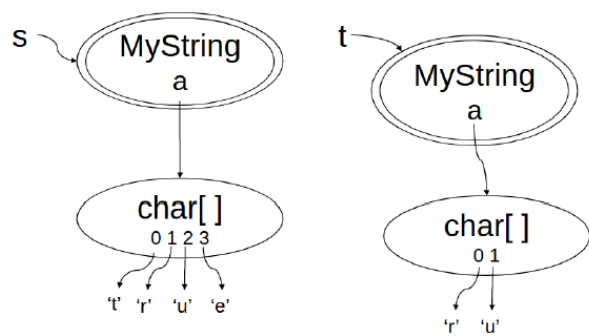
```



执行下列的代码

```
MyString s = MyString.valueOf(true);
MyString t = s.substring(1,3);
```

我们用快照图展示了在使用者进行 **subString** 操作后的数据状态：



这种实现有一个性能上的问题，因为这个数据类型是不可变的，那么 **substring** 实际上没有必要真正去复制子字符串到一个新的数组中。它可以仅仅指向原来的 **MyString** 字符数组，并且记录当前的起始位置和终止位置。

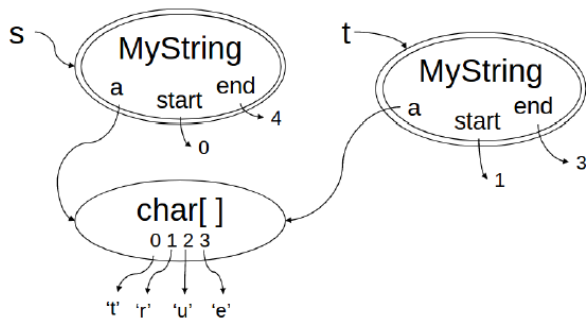
为了优化，我们可以将这个类的内部表示改为：

```
private char[] a;
private int start;
private int end;
```

有了这个新的表示，操作现在可以这样实现：

```
1 public static MyString valueOf(boolean b) {
2     MyString s = new MyString();
3     s.a = b ? new char[] { 't', 'r', 'u', 'e' }
4           : new char[] { 'f', 'a', 'l', 's', 'e' };
5     s.start = 0;
6     s.end = s.a.length;
7     return s;
8 }
9
10 public int length() {
11     return end - start;
12 }
13
14 public char charAt(int i) {
15     return a[start + i];
16 }
17
18 public MyString substring(int start, int end) {
19     MyString that = new MyString();
20     that.a = this.a;
21     that.start = this.start + start;
22     that.end = this.start + end;
23     return that;
24 }
```

现在运行上面的调用代码，可用快照图重新进行 **substring** 操作后的数据状态：



由于 `MyString` 的使用者仅依赖于其公共方法和规格说明，而不依赖其私有的存储，因此我们可以在不检查和更改所有客户端代码的情况下进行更改。这就是表示独立性的力量。

## ## 不变量 (Invariants) 与表示泄露

一个好的抽象数据类型的最重要的属性是它保持不变量。一旦一个不变类型的对象被创建，它总是代表一个不变的值。当一个ADT能够确保它内部的不变量恒定不变（不受使用者/外部影响），我们就说这个ADT保护/保留自己的不变量。

【一个栗子：表示泄露】

```
1 /**
2  * This immutable data type represents a tweet from Twitter.
3  */
4 public class Tweet {
5
6     public String author;
7     public String text;
8     public Date timestamp;
9
10    /**
11     * Make a Tweet.
12     * @param author    Twitter user who wrote the tweet
13     * @param text      text of the tweet
14     * @param timestamp date/time when the tweet was sent
15     */
16    public Tweet(String author, String text, Date timestamp) {
17        this.author = author;
18        this.text = text;
19        this.timestamp = timestamp;
20    }
21 }
```

我们如何保证这些 `Tweet` 对象是不可变的，（即一旦创建了 `Tweet`，其 `author`，`message` 和 `date` 永远不会改变）

对不可变性的第一个威胁来自使用者可以直接访问 `Tweet` 内部数据的事实，例如执行如下的引用操作：

```
1 Tweet t = new Tweet("justinbieber",
2                     "Thanks to all those believers out there inspiring me every day",
```


```

3             new Date());
4 t.author = "rbml1r";

```

这是一个表示泄露(**Rep exposure**)的简单例子，这意味着类外的代码可以直接修改表示。像这样的表示暴露不仅威胁到不变量，而且威胁到表示独立性。如果我们改变类内部数据的表示方式，使用者也会相应的受到影响。


幸运的是，**java**给我们提供了处理表示暴露的方法：



```

1 public class Tweet {
2     private final String author;
3     private final String text;
4     private final Date timestamp;
5
6     public Tweet(String author, String text, Date timestamp) {
7         this.author = author;
8         this.text = text;
9         this.timestamp = timestamp;
10    }
11
12    /** @return Twitter user who wrote the tweet */
13    public String getAuthor() {
14        return author;
15    }
16
17    /** @return text of the tweet */
18    public String getText() {
19        return text;
20    }
21
22    /** @return date/time when the tweet was sent */
23    public Date getTimestamp() {
24        return timestamp;
25    }
26 }

```



在**private**和**public**关键字表明哪些字段和方法可访问时，只在类内部还是可以从类外部访问。所述**final**关键字还保证该变量的索引不会被更改，对于不可变的类型来说，就是确保了变量的值不可变。

但这不能解决全部的问题：表示仍然会泄露！考虑这个完全合理的客户端代码，它使用**Tweet**：



```

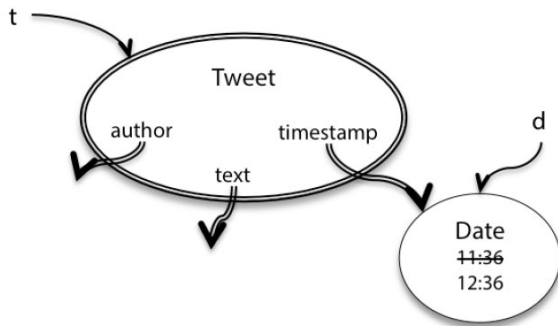
1 /** @return a tweet that retweets t, one hour later*/
2 public static Tweet retweetLater(Tweet t) {
3     Date d = t.getTimestamp();
4     d.setHours(d.getHours()+1);
5     return new Tweet("rbml1r", t.getText(), d);
6 }

```



retweetLater 希望接受一个Tweet对象然后修改Date后返回一个新的Tweet对象。

这里有什么问题？其中的 getTimestamp 调用返回一个一样的 Date 对象，它会被t、t.timestamp 和 d 同时索引。因此，当日期对象被突变，d.gsetHours() 被调用时，t 也会影响日期，如快照图所示。



这样，Tweet的不变性就被破坏，Tweet将自己内部对于可变对象的索引“泄露”了出来，因此整个对象都变成可变的了，使用者在使用时也容易造成隐藏的bug。

我们可以通过使用防御性拷贝来修补这种风险：制作可变对象的副本以避免泄漏对代表的引用。代码如下：

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

可变类型通常具有一个专门用来复制的构造函数，它允许创建一个复制现有实例值的新实例。在这种情况下，Date的复制构造函数就接受了一个timestamp值，然后产生一个新的对象。

复制可变对象的另一种方法是clone()，某些类型但不是全部类型支持该方法。然而clone()在Java中的工作方式存在问题，更多可参考[Effective Java, item 11](#)

现在我们已经通过防御性复制解决了 timestamp 返回值的问题。但我们还没有完成任务！还有表示泄露。考虑这个非常合理的客户端代码：

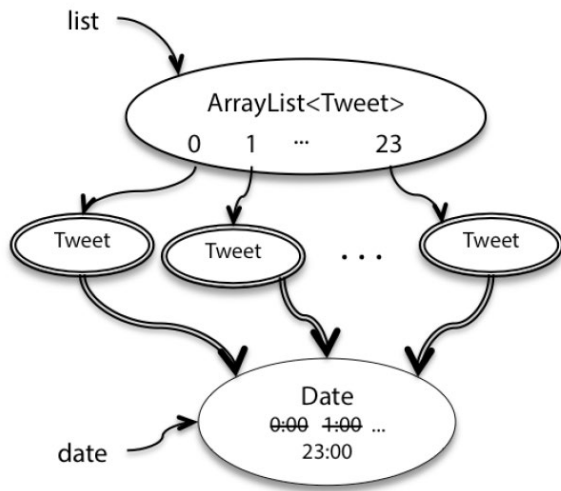


```
1 /** @return a list of 24 inspiring tweets, one per hour today */
2 public static List<Tweet> tweetEveryHourToday () {
3     List<Tweet> list = new ArrayList<Tweet>();
4     Date date = new Date();
5     for (int i = 0; i < 24; i++) {
6         date.setHours(i);
7         list.add(new Tweet("rbmlr", "keep it up! you can do it", date));
8     }
9     return list;
10 }
```





此代码旨在创建24个Tweet对象，为每个小时创建一条推文。但请注意，Tweet的构造函数保存传入的引用，因此所有24个Tweet对象最终都以同一时间结束，如此快照图所示。



但是，Tweet的不变性再次被打破了，因为每一个Tweet创建时对Date对象的索引都是一样的。所以我们应该对创建者也进行防御性编程：

```
1 public Tweet(String author, String text, Date timestamp) {
2     this.author = author;
3     this.text = text;
4     this.timestamp = new Date(timestamp.getTime());
5 }
```

通常来说，要特别注意ADT操作中的参数和返回值。如果它们之中有可变类型的对象，确保你的代码没有直接使用索引或者直接返回索引。

你可能反对说这看起来很浪费。为什么要制作所有这些日期的副本？为什么我们不能通过像这样**仔细书写的规范**来解决这个问题？

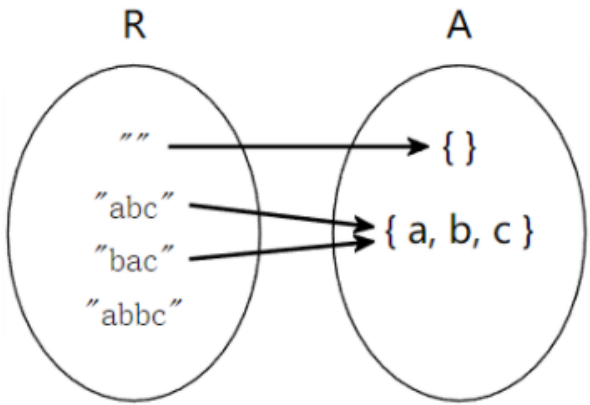
```
/**
 * Make a Tweet.
 * @param author    Twitter user who wrote the tweet
 * @param text      text of the tweet
 * @param timestamp date/time when the tweet was sent. Caller must never
 *                  mutate this Date object again!
 */
public Tweet(String author, String text, Date timestamp)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

这种方法一般只在不得已的时候使用——例如，当可变对象太大而无法有效地复制时。但是，由此引发的潜在bug也将很多。除非迫不得已，否则不要把希望寄托于客户端上，ADT有责任保证自己的不变量，并避免表示泄露。

最好的办法就是使用immutable的类型，彻底避免表示泄露，例如 `java.time.ZonedDateTime` 而不是 `java.util.Date`。

## 抽象函数AF与表示不变量RI

【AF与RI】



- 在研究抽象类型的时候，先思考一下两个值域之间的关系：
  - 表示域（rep values）里面包含的是值具体的实现实体。一般情况下ADT的表示比较简单，有些时候需要复杂表示。
  - 抽象域（A）里面包含的则是类型设计时支持使用的值。这些值是由表示域“抽象/想象”出来的，也是使用者关注的。
- ADT实现者关注表示空间R，用户关注抽象空间A。
- R->A的映射特点：
  - 每一个抽象值都是由表示值映射而来，即满射：每个抽象值被映射到一些rep值
  - 一些抽象值是被多个表示值映射而来的，即未必单射：一些抽象值被映射到多个rep值
  - 不是所有的表示值都能映射到抽象域中，即未必双射：并非所有的rep值都被映射。

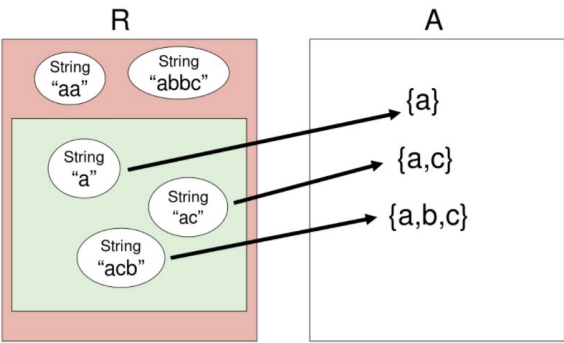
抽象函数（AF）：R和A之间映射关系的函数

```
1 AF : R → A
```

表示不变量（RI）：将rep值映射到布尔值

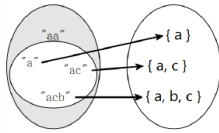
```
1 RI : R → boolean
```

- 对于表示值r，当且仅当r被AF映射到了A，RI(r)为真。
- 表示不变性RI：某个具体的“表示”是否是“合法的”
- 也可将RI看作：所有表示值的一个子集，包含了所有合法的表示值
- 也可将RI看作：一个条件，描述了什么是“合法”的表示值
- 在下图中，绿色表示的就是RI(r)为真的部分，AF只在这个子集上有定义。



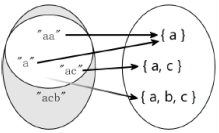
表示不变量和抽象函数都应该记录在代码中，就在代表本身的声明旁边，以下图为例

```
public class CharSet {
    private String s;
    // Rep invariant:
    //   s contains no repeated characters
    // Abstraction Function:
    //   represents the set of characters found in s
    ...
}
```



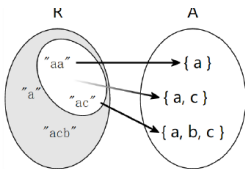
```
public class CharSet {
    private String s;
    // Rep invariant:
    //   s contains no repeated characters
    // Abstraction function:
    //   AF(s) = {s[i] | 0 <= i < s.length()}
    ...
}
```

```
public class CharSet {
    private String s;
    // Rep invariant:
    //   s[0] <= s[1] <= ... <= s[s.length()-1]
    // Abstraction Function:
    //   represents the set of characters found in s
    ...
}
```



```
public class CharSet {
    private String s;
    // Rep invariant:
    //   s[0] <= s[1] <= ... <= s[s.length()-1]
    // Abstraction function:
    //   AF(s) = {s[i] | 0 <= i < s.length()}
    ...
}
```

```
public class CharSet {
    private String s;
    // Rep invariant:
    //   s.length is even
    //   s[0] <= s[1] <= ... <= s[s.length()-1]
    // Abstraction Function:
    //   represents the union of the ranges
    //   {s[i]...s[i+1]} for each adjacent pair of characters
    //   in s
    ...
}
```



```
public class CharSet {
    private String s;
    // Rep invariant:
    //   s.length() is even
    //   s[0] <= s[1] <= ... <= s[s.length()-1]
    // Abstraction function:
    //   AF(s) = union of {s[2i],...,s[2i+1]} for 0 <= i < s.length()/2
    ...
}
```

```
...
}
```



### 【用注释写AF和RI】

- 在抽象类型（私有的）表示声明后写上对于抽象函数和表示不变量的注解，这是一个好的实践要求。我们在上面的例子中也是这么做的。
- 在描述抽象函数和表示不变量的时候，注意要清晰明确：
  - 对于RI（表示不变量），仅仅宽泛的说什么区域是合法的并不够，你还应该说明是什么使得它合法/不合法。
  - 对于AF（抽象函数）来说，仅仅宽泛的说抽象域表示了什么并不够。抽象函数的作用是规定合法的表示值会如何被解译到抽象域。作为一个函数，我们应该清晰的知道从一个输入到一个输入是怎么对应的。
- 本门课程还要求你将表示暴露的安全性注释出来。这种注释应该说明表示的每一部分，它们为什么不会发生表示暴露，特别是处理的表示的参数输入和返回部分（这也是表示暴露发生的位置）。
- 下面是一个Tweet类的例子，它将表示不变量和抽象函数以及表示暴露的安全性注释了出来：



```
1 // Immutable type representing a tweet.
2 public class Tweet {
3
4     private final String author;
5     private final String text;
6     private final Date timestamp;
7
8     // Rep invariant:
9     //   author is a Twitter username (a nonempty string of letters, digits, underscores)
10    //   text.length <= 140
11    // Abstraction function:
12    //   AF(author, text, timestamp) = a tweet posted by author, with content text,
13    //                                   at time timestamp
14    // Safety from rep exposure:
15    //   All fields are private;
16    //   author and text are Strings, so are guaranteed immutable;
17    //   timestamp is a mutable Date, so Tweet() constructor and getTimestamp()
18    //       make defensive copies to avoid sharing the rep's Date object with clients.
19
20    // Operations (specs and method bodies omitted to save space)
21    public Tweet(String author, String text, Date timestamp) { ... }
22    public String getAuthor() { ... }
23    public String getText() { ... }
24    public Date getTimestamp() { ... }
25 }
```



注意到我们并没有对 timestamp 的表示不变量进行要求（除了之前说过的默认 timestamp!=null）。但是我们依然需要对timestamp 的表示暴露的安全性进行说明，因为整个类型的不变性依赖于所有的成员变量的不变性。