哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

6 Abstract Data Type (ADT)

抽象数据类型 (ADT)

Wang Zhongjie
rainy@hit.edu.cn

March 24, 2024

Outline

1. Abstraction and User-Defined Types
2. Classification of operations in ADT
3. Abstract Data Type Examples
4. Design principles of ADT
5. Representation Independence (RI)
6. Realizing ADT Concepts in Java
7. Testing an ADT
8. Invariants
9. Rep Invariant and Abstraction Function
10. Beneficent mutation
11. Documenting the AF, RI, and Safety from Rep Exposure
12. ADT invariants replace preconditions

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

本节：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

Objective of this lecture

- **Abstract data types and representation independence** : enable us to separate how we use a data structure in a program from the particular form of the data structure itself.
  - Abstract data types address a particularly dangerous problem: clients making assumptions about the type's internal representation.
  - We'll see why this is dangerous and how it can be avoided.
  - We'll also discuss the classification of operations, and some principles of good design for abstract data types.

- **抽象数据类型与表示独立性：如何设计良好的抽象数据结构，通过封装来避免客户端获取数据的内部表示（即“表示泄露”），避免潜在的bug——在client和implementer之间建立“防火墙”**

Objective of this lecture

- **Invariants, representation exposure, abstraction functions (AF), and representation invariants (RI)**
  - A more formal mathematical idea of what it means for a class to implement an ADT, via the notions of *abstraction functions* and *rep invariants* .
  - These mathematical notions are eminently practical in software design.
  - The abstraction function will give us a way to cleanly define the equality operation on an abstract data type.
  - The rep invariant will make it easier to catch bugs caused by a corrupted data structure.

- **ADT的特性：表示泄漏、抽象函数AF、表示不变量RI**
- **基于数学的形式对ADT的这些核心特征进行描述并应用于设计中。**

Reading

- **MIT 6.031：10、11**

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

1 Abstraction and User-Defined Types

User-Defined Types

- A programming language came with built-in types (such as `integers`, `booleans`, `strings`, etc.) and built-in procedures, e.g., for input and output.

- Users could define their own data types and procedures – **User-Defined Types**.

- 除了编程语言所提供的基本数据类型和对象数据类型，程序员可定义自己的数据类型

Data Abstraction

- **Data abstraction: a type is characterized by the operations you can perform on it.** 数据抽象：由一组操作所刻画的数据类型（而非...）
  - A number is something you can add and multiply;
  - A string is something you can concatenate and take substrings of;
- Traditionally, programmers define their own types in early programming languages, such as: create a record type `date`, with `integer` fields for day, month, and year. 传统的类型定义：关注数据的具体表示
- Abstract types focus on **operations** --- user of the type need not to worry about how its values were actually stored, a programmer can ignore how the compiler actually stores integers. All that matters is the **operations**. 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。

An example

- **An abstract data type `Bool` has the following operations:**
  - `true:` `Bool`
  - `false:` `Bool`
  - `and:` `Bool × Bool → Bool`
  - `or:` `Bool × Bool → Bool`
  - `not:` `Bool → Bool`

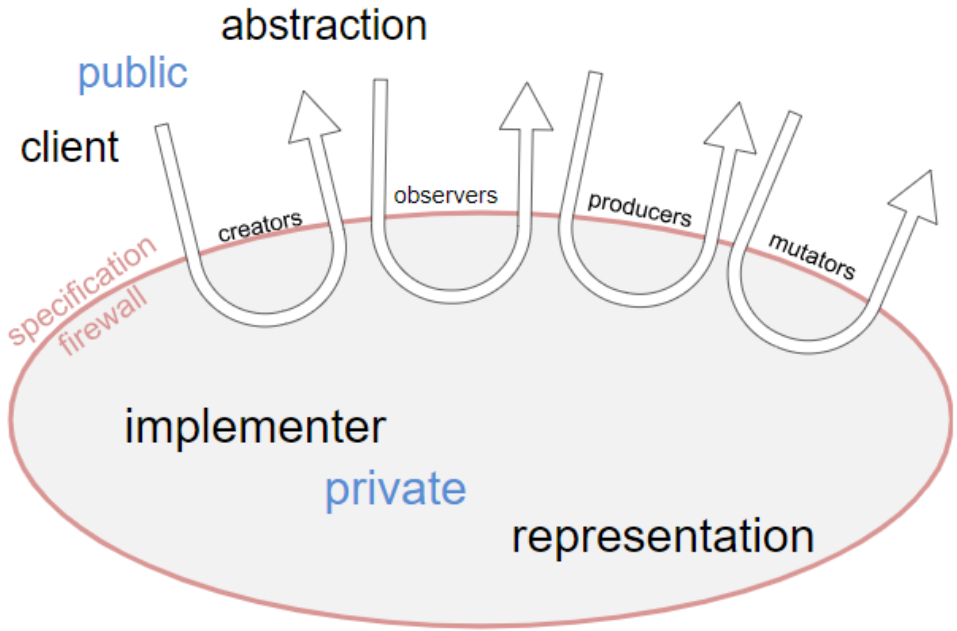
The operations themselves (and their specs) completely define the data type, abstracting away from the details of data structure, memory storage, or implementation.

- **There are many possible ways that `Bool` might be implemented, and still be able to satisfy the specs of the operations, for example:**
  - As a single `bit`, where 1 means true and 0 means false.
  - As an `int` value where 5 means true and 8 means false.
  - As a reference to a `String` object where "false" means true and "true" means false.
  - As an `int` value > 1 where prime numbers mean true and composite numbers mean false.

An abstract type is defined by its operations

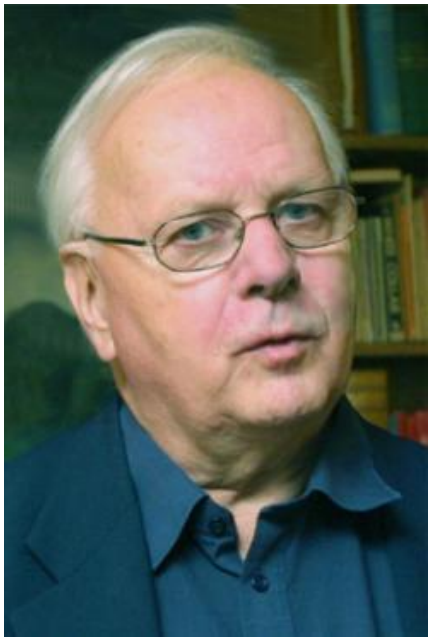
- **Remember----an abstract data type is defined by its operations.**
  - The set of operations for a type T, along with their specifications, fully characterize what we mean by T.
- **When we talk about the `List` type, what we mean is not a `linked list` or an `array` or any other specific data structure for representing a list.**
- **Instead we mean a set of opaque values – the possible objects that can have `List` type – that satisfy the specifications of all the operations of `List`: `get()`, `size()`, etc.**

ADT是由操作定义的，与其内部如何实现无关！



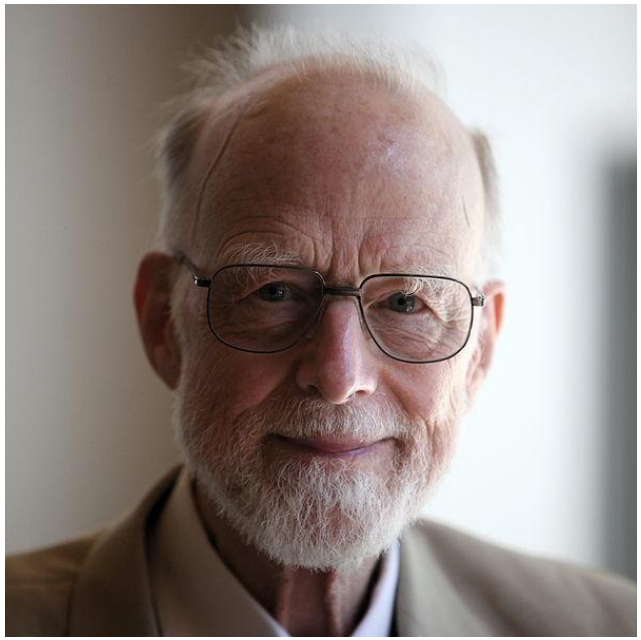
SE scholars who made contributions to ADT

- **Ole-Johan Dahl** and **Kristen Nygaard Dahl** (the inventors of the Simula language)
- **Antony Hoare** (who developed many of the techniques we now use to reason about abstract types)



Kristen Nygaard (1926-2002)
2001 Turing Award

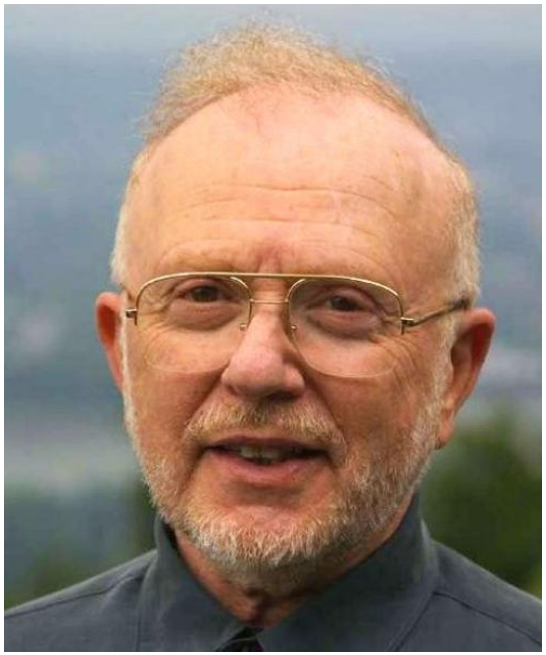
Ole-Johan Dahl (1931-2002)
2001 Turing Award



Antony Hoare (1934-)
1980 Turing Award

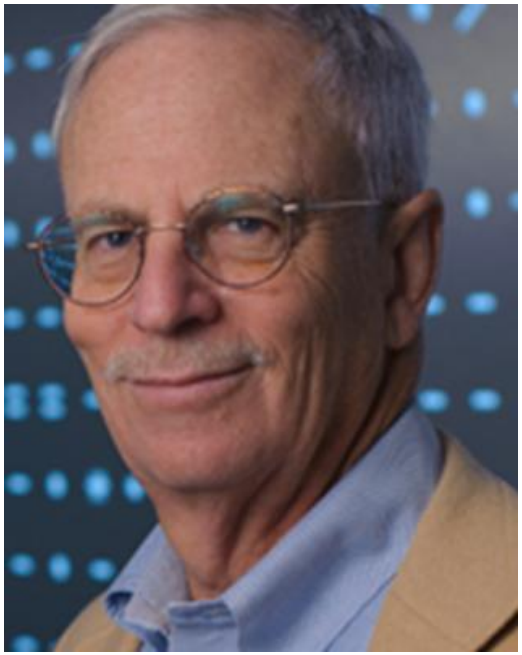
SE scholars who made contributions to ADT

- **David Parnas** (who coined the term information hiding and first articulated the idea of organizing program modules around the secrets they encapsulated)
- **Barbara Liskov** and **John Guttag** (the specification of abstract types, and in programming language support for them)



David L. Parnas (1941-)

Barbara Liskov (1939-)
2008 Turing Award



John Guttag (1949-)

2 Classifying Types and Operations

Mutable and immutable types

- **Types, whether built-in or user-defined, can be classified as mutable or immutable** 可变和不可变数据类型
  - The objects of a mutable type can be changed: that is, they provide operations which when executed cause the results of other operations on the same object to give different results. 可变类型的对象：提供了可改变其内部数据的值的操作
  - `Date` is mutable, because you can call `setMonth()` and observe the change with the `getMonth()` operation.
  - But `String` is immutable, because its operations create new `String` objects rather than changing existing ones. 不变数据类型： 其操作不改变内部值，而是构造新的对象
  - Sometimes a type will be provided in two forms, a mutable and an immutable form. `StringBuilder`, for example, is a mutable version of `String` (but the e two are certainly not the same Java type, and are not interchangeable).

Classifying the operations of an abstract type

- **Creators** create new objects of the type. 构造器
  - A creator may take an object as an argument, but not an object of the type being constructed.
- **Producers** create new objects from old objects of the type. 生产器
  - The `concat()` method of `String` , for example, is a producer: it takes two strings and produces a new one representing their concatenation.
- **Observers** take objects of the abstract type and return objects of a different type. 观察器
  - The `size()` method of `List` , for example, returns an `int` .
- **Mutators** change objects. 变值器，改变对象属性的方法
  - The `add()` method of `List` , for example, mutates a list by adding an element to the end.

Classifying the operations of an abstract type

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

- **Each `T` is the abstract type itself;**
- **Each `t` is some other type.**
- **The + marker indicates that the type may occur one or more times in that part of the signature.**
- **The * marker indicates that it occurs zero or more times.**
- **The | indicates or.**

Signature of an operation

- `String.concat()` **as a producer**
  - `concat: String × String → String`
- `List.size()` **as an observer**
  - `size: List → int`
- `String.regionMatches` **as a observer**
  - `regionMatches:`
    `String × boolean × int × String × int × int → boolean`

regionMatches

```
public boolean regionMatches(boolean ignoreCase,
                             int toffset,
                             String other,
                             int ooffset,
                             int len)
```

Tests if two string regions are equal.

请阅读该函数的spec，学习复杂spec的写法

Signature of a creator

- A creator is either implemented as a constructor , like `new ArrayList()`, or simply a static method instead, like `Arrays.asList(), List.of()`. 构造器：可能实现为构造函数或静态函数

- A creator implemented as a static method is often called a factory method 工厂方法

- The various `String.valueOf(Object Obj)` methods in Java are other examples of creators implemented as factory methods. 与`Object.toString()`正好是相对的

Signature of a mutator

- **Mutators are often signaled by a `void` return type.** 变值器通常返回 **void**
  - A method that returns `void` must be called for some kind of side-effect, since otherwise it doesn't return anything. 如果返回值为`void`，则必然意味着它改变了对象的某些内部状态

- **But not all mutators return `void`.** 变值器也可能返回非空类型
  - For example, `Set.add()` returns a `boolean` that indicates whether the set was actually changed.
  - In Java's graphical user interface toolkit, `Component.add()` returns the object itself, so that multiple `add()` calls can be chained together.

3 Abstract Data Type Examples

`int` and `String`

- **`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

List

https://docs.oracle.com/javase/8/docs/api/java/util/List.html

看看每一个method的类型是什么

- **`List` is Java's list type and is mutable.**
- **`List` is also an interface, which means that other classes provide the actual implementation of the data type, such as `ArrayList` and `LinkedList`.**
  - creators: `ArrayList` and `LinkedList` constructors, `Collections.singletonList`
  - producers: `Collections.unmodifiableList`
  - observers: `size`, `get`
  - mutators: `add`, `remove`, `addAll`, `Collections.sort`

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
boolean	**`add(E e)`** Appends the specified element to the end of this list (optional operation).
void	**`add(int index, E element)`** Inserts the specified element at the specified position in this list (optional operation).
boolean	**`addAll(Collection<? extends E> c)`** Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	**`addAll(int index, Collection<? extends E> c)`** Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	**`clear()`** Removes all of the elements from this list (optional operation).
boolean	**`contains(Object o)`** Returns `true` if this list contains the specified element.
boolean	**`containsAll(Collection<?> c)`** Returns `true` if this list contains all of the elements of the specified collection.

Examples

`Integer.valueOf()`	**Creator**
`BigInteger.mod()`	**Producer**
`List.addAll()`	**Mutator**
`String.toUpperCase()`	**Producer**
`Set.contains()`	**Observer**
`Map.keySet()`	**Observer**
`Collections.unmodifiableList()`	**Producer**
`BufferedReader.readLine()`	**Mutator**

4 Designing an Abstract Type

Designing an Abstract Type

- **Designing an abstract type involves choosing good operations and determining how they should behave. 设计好的ADT，靠“经验法则”，提供一组操作，设计其行为规约 spec**

- **Rules of thumb 1 设计简洁、一致的操作**
  - It’s better to have **a few, simple operations** that can be combined in powerful ways, rather than lots of complex operations.
  - Each operation should have a **well-defined purpose**, and should have a **coherent behavior** rather than a panoply of special cases.
  - We probably shouldn’t add a `sum` operation to `List` , for example. It might help clients who work with lists of integers, but what about lists of strings? Or nested lists? All these special cases would make `sum` a hard operation to understand and use.

Designing an Abstract Type

- **Rules of thumb 2 要足以支持client对数据所做的所有操作需要，且用操作满足client需要的难度要低**
- **The set of operations should be adequate in the sense that there must be enough to do the kinds of computations clients are likely to want to do.**
  - A good test is to check that every property of an object of the type can be extracted.
  - For example, if there were no `get` operation, we would not be able to find out what the elements of a list are. 没有`get()`操作就无法获取`list`的内部数据
  - Basic information should not be inordinately difficult to obtain.
  - For example, the size method is not strictly necessary for `List`, because we could apply get on increasing indices until we get a failure, but this is inefficient and inconvenient. 用遍历方式获取`list`的`size`-太复杂 vs 提供`size()`操作，方便client使用

Designing an Abstract Type

- **Rules of thumb 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**

- **The type may be generic: a list or a set, or a graph, for example.**
- **Or it may be domain-specific: a street map, an employee database, a phone book, etc.**

- **But it should not mix generic and domain-specific features.**
  - A `Deck` type intended to represent a sequence of playing cards shouldn't have a generic `add` method that accepts arbitrary objects like `integers` or `strings`.
  - Conversely, it wouldn't make sense to put a domain-specific method like `dealCards` into the generic type `List` .

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

5 Representation Independence

表示独立性

Representation Independence

- Critically, a good abstract data type should be **representation independent.** **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。**
  - The use of an abstract type is independent of its representation (the actual data structure or data fields used to implement it), so that changes in representation have no effect on code outside the abstract type itself.
  - For example, the operations offered by `List` are independent of whether the list is represented as a `linked list` or as an `array`.

- You won't be able to change the representation of an ADT at all unless its operations are fully specified with preconditions and postconditions, so that clients know what to depend on, and you know what you can safely change. **除非ADT的操作指明了具体的pre-和post-condition，否则不能改变ADT的内部表示——spec规定了client和implementer之间的契约。**

Example: Different Representations for Strings

```
/** 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() { ... }

    /** @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) { ... }
}
```

Let's write its test case (Test-First Programming)

```
MyString s = MyString.valueOf(true);
assertEquals("true", s);
assertEquals("true", s.toString());

assertEquals(4, s.length());

assertEquals('t', s.charAt(0));
assertEquals('r', s.charAt(1));
assertEquals('u', s.charAt(2));
assertEquals('e', s.charAt(3));

MyString t = s.substring(0,2);
assertEquals('t', t.charAt(0));
...
```

A simple representation for `MyString`

- **Look at a simple representation for `MyString` : just an array of characters, exactly the length of the string, with no extra room at the end.**
- **Here's how that internal representation would be declared, as an instance variable within the class:**

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

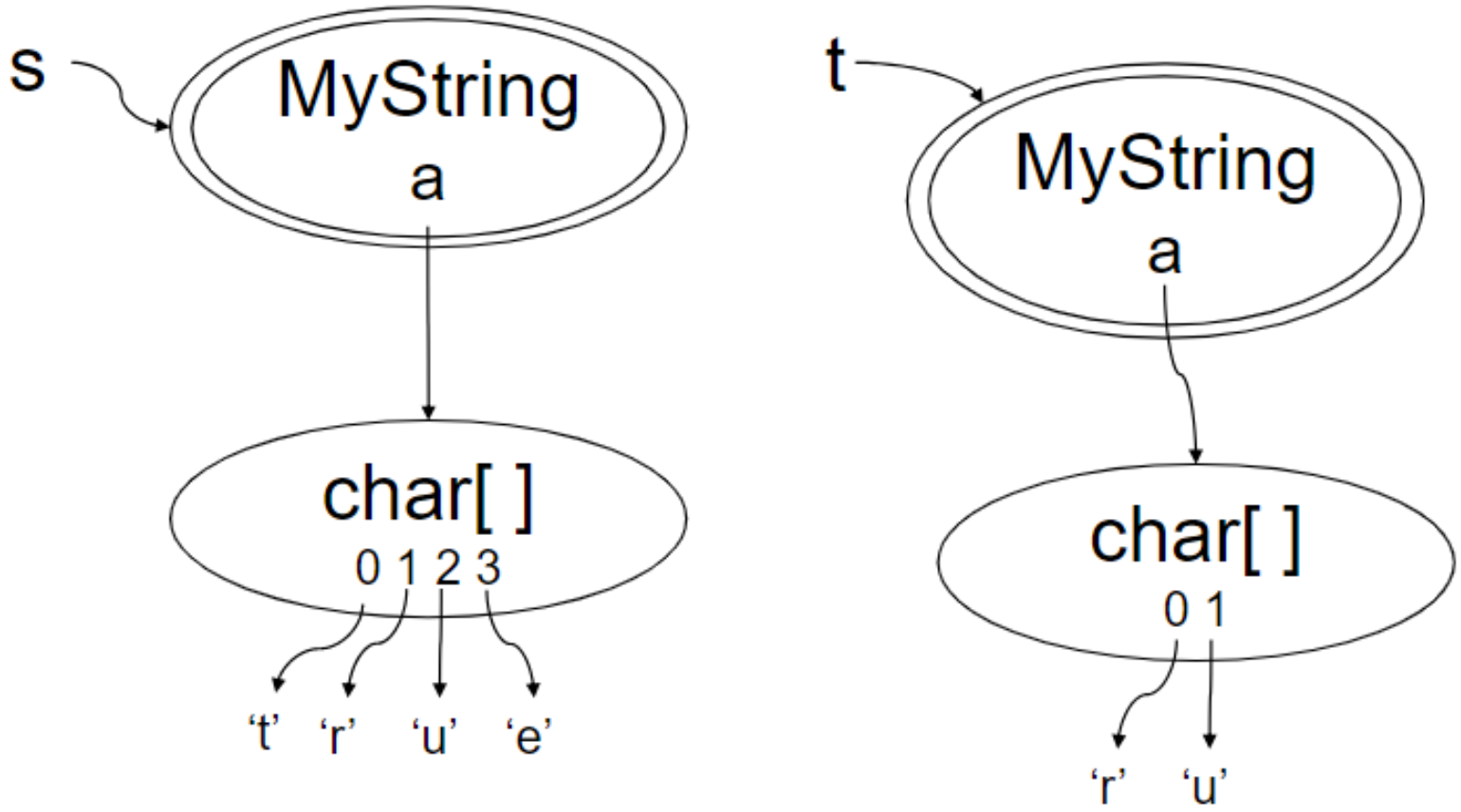
- **With that choice of representation, the operations would be implemented in a straightforward way:**

The corresponding implementation for `MyString`

```
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;
}
```

An example of the client

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



Another representation for better performance

- **Because this data type is immutable, the `substring` operation doesn't really have to copy characters out into a fresh array.**
- **It could just point to the original `MyString` object's character array and keep track of the start and end that the new substring object represents.**
- **To implement this optimization, we could change the internal representation of this class to:**

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

Now the implementation is …

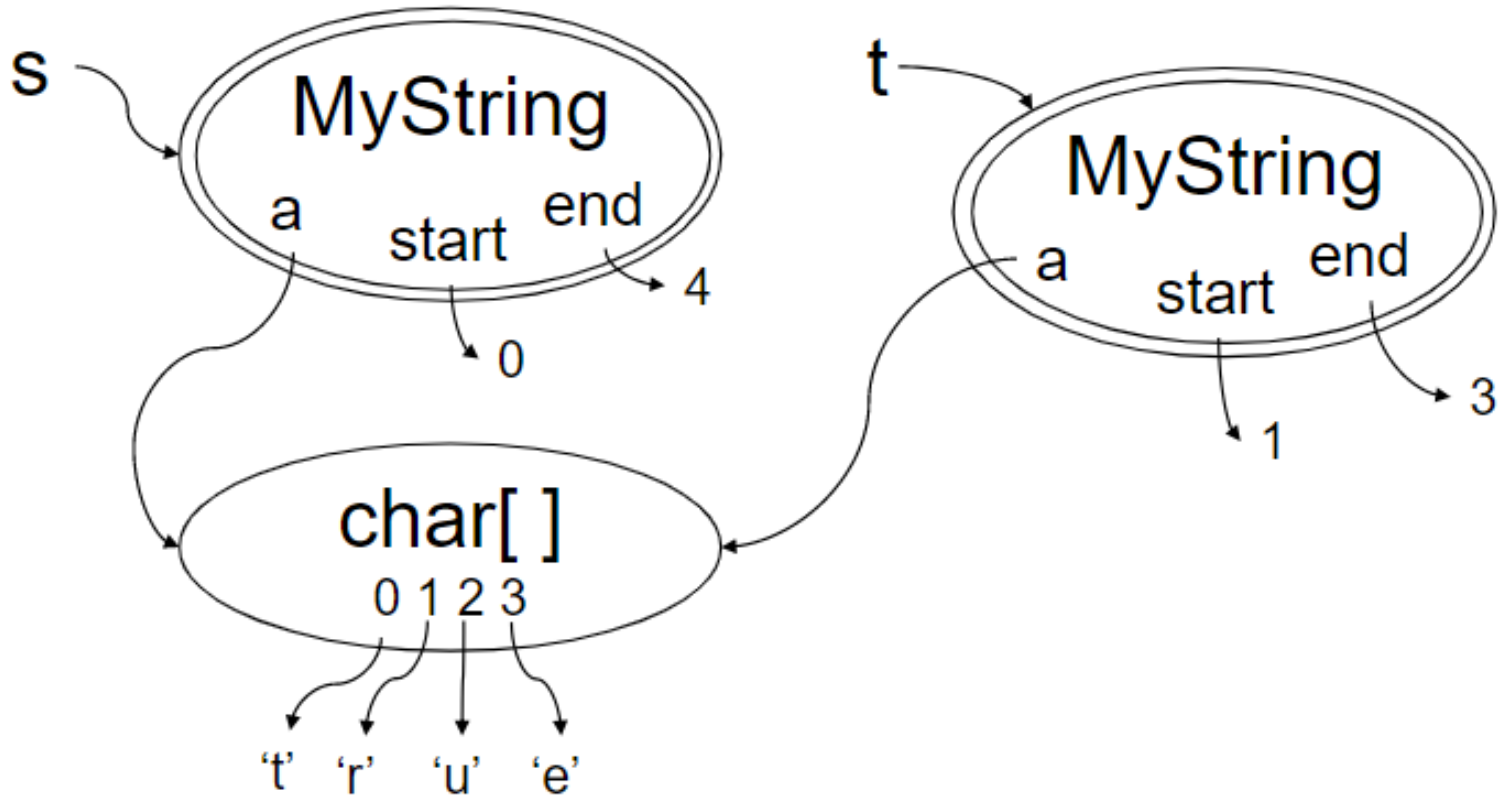
```
public static MyString valueOf(boolean b) {
    MyString s = new MyString();
    s.a = b ? new char[] { 't', 'r', 'u', 'e' }
            : new char[] { 'f', 'a', 'l', 's', 'e' };
    s.start = 0;
    s.end = s.a.length;
    return s;
}

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;
}
```

Because `MyString`'s existing clients depend only on the specs of its public methods, not on its private fields, we can make this change without having to inspect and change all that client code.

That's the power of representation independence.

An example of the client

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



An example of violating RI

```
/**
 * Represents a family that lives in a household together.
 * A family always has at least one person in it.
 * Families are mutable.
 */
class Family {
    public List<Person> people;
    public List<Person> getMembers() {
        return people;
    }
}
```

```
class Family {
  Set<Person> p;
  List<Person> getMembers() {
    return new ArrayList<>(p);
  }
}
```

```
void client1(Family f) {
    Person baby = f.people.get(f.people.size()-1); ...
}
```

An example of keeping RI

Original version:

```
/**
 * Represents a family that lives in a
 * household together. A family always
 * has at least one person in it.
 * Families are mutable. */
class Family {
    // the people in the family,
    // sorted from oldest to youngest,
    // with no duplicates.
    public List<Person> people;

    /** @return a list containing all
     *  the members of the family,
     *  with no duplicates. */
    public List<Person> getMembers() {
        return people;
    }
}
```

Changed version:

```
/**
 * Represents a family that lives in a
 * household together. A family always
 * has at least one person in it.
 * Families are mutable. */
class Family {
    // the people in the family
    public Set<Person> people;


    /**
     * @return a list containing all
     * the members of the family,
     * with no duplicates. */
    public List<Person> getMembers() {
        return new ArrayList<>(people);
    }
```

```
void client3(Family f) {
    // get any person in the family
    Person anybody = f.getMembers().get(0);
    ...
}
```

Rep ?

An example of keeping RI

```java
/**
 * Represents a family that lives in a household together.
 * A family always has at least one person in it.
 * Families are mutable.
 */
class Family {
    // the people in the family, sorted from oldest to youngest, with no duplicates.
    public List<Person> people;

    /**
     * @return a list containing all the members of the family, with no duplicates.
     */
    public List<Person> getMembers() {
        return people;
    }
}
```

Specification Representation Implementation

An example of keeping RI

```
/**
 * Represents a family that lives in a household together.
 * A family always has at least one person in it.
 * Families are mutable.
 */
class Family {
    // the people in the family, sorted from oldest to youngest, with no duplicates.
    public List<Person> people;

    /**
     * @return a list containing all the members of the family, with no duplicates.
     */
    public List<Person> getMembers() {
        return people;
    }
}
```

Specification

Specification

Representation

Specification

Implementation

6 Testing an Abstract Data Type

How to test an ADT

- We build a test suite for an ADT by creating tests for each of its operations.
- These tests inevitably interact with each other.
- The only way to test creators, producers, and mutators is by calling observers on the objects that result, and likewise, the only way to test observers is by creating objects for them to observe.

- **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。
- **风险：如果被依赖的其他方法有错误，可能导致被测试方法的测试结果失效。**

Partition the input spaces of ADT operations

```
// testing strategy for each operation of MyString:
//
// valueOf():
//    true, false
// length():
//    string len = 0, 1, n
//    string = produced by valueOf(), produced by substring()
// charAt():
//    string len = 1, n
//    i = 0, middle, len-1
//    string = produced by valueOf(), produced by substring()
// substring():
//    string len = 0, 1, n
//    start = 0, middle, len
//    end = 0, middle, len
//    end-start = 0, n
//    string = produced by valueOf(), produced by substring()
```

针对creator：构造对象之后，用observer去观察是否正确

针对observer：用其他三类方法构造对象，然后调用被测observer，判断观察结果是否正确

针对producer：produce新对象之后，用observer判断结果是否正确

Recall what we've learned in Section 7-5: partitioning-based testing

Test suite that covers all partitions

- Which test cases cover the part "`charAt()` with string `len=1`"?
- Which test cases cover the part "`substring()` of string produced by `substring()`"?
- Which test cases cover the part "`valueOf(true)`"?

```
@Test public void testValueOfTrue() {
    MyString s = MyString.valueOf(true);
    assertEquals(4, s.length());
    assertEquals('t', s.charAt(0));
    assertEquals('r', s.charAt(1));
    assertEquals('u', s.charAt(2));
    assertEquals('e', s.charAt(3));
}

@Test public void testValueOfFalse() {
    MyString s = MyString.valueOf(false);
    assertEquals(5, s.length());
    assertEquals('f', s.charAt(0));
    assertEquals('a', s.charAt(1));
    assertEquals('l', s.charAt(2));
    assertEquals('s', s.charAt(3));
    assertEquals('e', s.charAt(4));
}

@Test public void testEndSubstring() {
    MyString s = MyString.valueOf(true).substring(2, 4);
    assertEquals(2, s.length());
    assertEquals('u', s.charAt(0));
    assertEquals('e', s.charAt(1));
}
```

```
@Test public void testMiddleSubstring() {
    MyString s = MyString.valueOf(false).substring(1, 2);
    assertEquals(1, s.length());
    assertEquals('a', s.charAt(0));
}

@Test public void testSubstringIsWholeString() {
    MyString s = MyString.valueOf(false).substring(0, 5);
    assertEquals(5, s.length());
    assertEquals('f', s.charAt(0));
    assertEquals('a', s.charAt(1));
    assertEquals('l', s.charAt(2));
    assertEquals('s', s.charAt(3));
    assertEquals('e', s.charAt(4));
}

@Test public void testSubstringOfEmptySubstring() {
    MyString s = MyString.valueOf(false).substring(1, 1).substring(0, 0);
    assertEquals(0, s.length());
}
```

7 Invariants

表示不变量:

- 这是对象内部状态必须始终满足的一个或一组条件，用以保持对象的一致性和有效性。
- 表示不变量定义了对象的合法状态。例如，对于一个表示分数的对象，表示不变量可能是分母不为零。
- 它是类的内部实现的一部分，确保即使在对象的公开方法被调用之后，对象仍然处于有效状态。
- 表示不变量是面向对象编程中封装概念的一部分，通常在设计和实现阶段得到强调和维护。

表示独立性:

- 它指的是一个组件的内部实现（如数据的表示和存储结构）可以更改，而不影响依赖该组件的外部代码。
- 表示独立性允许开发者更改对象的内部结构或者算法，而不需要修改依赖于这个对象接口的代码。
- 这与抽象化和信息隐藏原则紧密相连，因为客户端代码只依赖于抽象接口而不是具体实现。
- 它有助于保持代码的可维护性和可扩展性。

Invariants of an ADT

- The most important property of a good abstract data type is that it **preserves its own invariants**. 保持不变量
- An *invariant* is a property of a program that is always true, for every possible runtime state of the program. 不变量：在任何时候总是true
  - Immutability is one crucial invariant: once created, an immutable object should always represent the same value, for its entire lifetime. 例如：immutability就是一个典型的“不变量”
- Saying that the **ADT *preserves its own invariants*** means that the ADT is responsible for ensuring that its own invariants hold. 由ADT来负责其不变量，与client端的任何行为无关
  - It doesn’t depend on good behavior from its clients.
  - Correctness doesn’t depend on other modules.

Why are invariants required?

- **When an ADT preserves its own invariants, reasoning about the code becomes much easier.** 为什么需要不变量：保持程序的“正确性”，容易发现错误
  - If you can count on the fact that Strings never change, you can rule out that possibility when you're debugging code that uses Strings – or when you're trying to establish an invariant for another ADT that uses Strings.
  - Contrast that with a string type that guarantees that it will be immutable only if its clients promise not to change it. Then you'd have to check all the places in the code where the string might be used. 如果没有这个不变性，那么在所有使用String的地方，都要检查其是否改变了

- **… Assume clients will try to destroy invariants (malicious hackers or honest mistakes)** **总是要假设client有“恶意”破坏ADT的不变量---defensive programming**

Immutability as a type of Invariants

- How do we guarantee that these `Tweet` objects are immutable – that, once a tweet is created, its author, message, and date can never be changed?

```
/**
 * This immutable data type represents a tweet from Twitter.
 */
public class Tweet {

    public String author;
    public String text;
    public Date timestamp;

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

It's mutable…

- The first threat to immutability comes from the fact that clients can directly access its fields.
- **What's the effect of this code?**

```
Tweet t = new Tweet("justinbieber",
                    "Thanks to all those beliebers out there inspiring me every day",
                    new Date());
t.author = "rbmllr";
```

- This is a trivial example of **representation exposure** 表示泄露, meaning that code outside the class can modify the representation directly. 不仅影响不变性，也影响了表示独立性：无法在不影响客户端的情况下改变其内部表示
  - Rep exposure like this threatens not only invariants, but also representation independence.
  - We can't change the implementation of `Tweet` without affecting all the clients who are directly accessing those fields.

To make it immutable…

- The `private` and `public` keywords indicate which fields and methods are accessible only within the class and which can be accessed from outside the class.
- The `final` keyword also helps by guaranteeing that the fields of this immutable type won't be reassigned after the object is constructed.

```
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;
    }

    /** @return Twitter user who wrote the tweet */
    public String getAuthor() {
        return author;
    }

    /** @return text of the tweet */
    public String getText() {
        return text;
    }

    /** @return date/time when the tweet was sent */
    public Date getTimestamp() {
        return timestamp;
    }

}
```

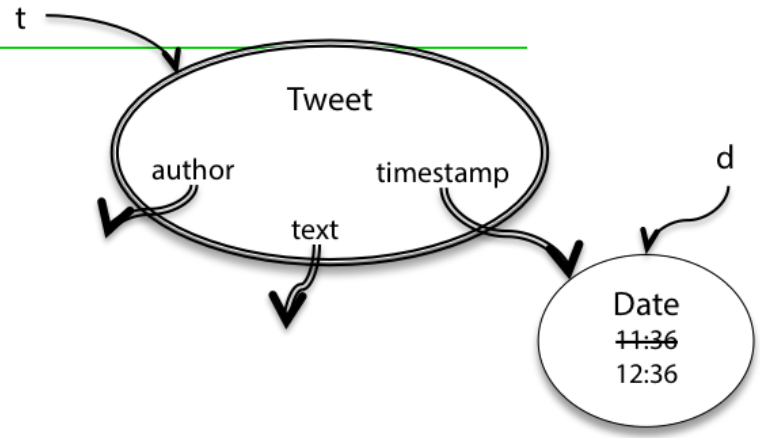
How about this …

- **What's the effect of this code?**

```
/** @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);
}
```

- **`retweetLater()` takes a tweet and should return another tweet with the same message (called a retweet) but sent an hour later.**
- **The `retweetLater()` method might be part of a system that automatically echoes funny things that Twitter celebrities say.**

What's the problem?



- **What's the problem here?**
  - The `getTimestamp` call returns a reference to the same `Date` object referenced by tweet `t` . `t.timestamp` and `d` are aliases to the same mutable object.
  - So when that date object is mutated by `d.setHours()` , this affects the date in `t` as well, as shown in the snapshot diagram.

But, we have declared `timestamp` as `final`!

- **Tweet 's immutability invariant has been broken.**
  - The problem is that `Tweet` leaked out a reference to a mutable object that its immutability depended on.
  - We exposed the rep, in such a way that `Tweet` can no longer guarantee that its objects are immutable.
  - Perfectly reasonable client code created a subtle bug.

How to solve it? --- Defensive copying

- **We can patch this kind of rep exposure by using defensive copying: making a copy of a mutable object to avoid leaking out references to the rep.**

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

- **Defensive copying is an approach of defensive programming**
  - Assume clients will try to destroy invariants --- May actually be true (malicious hackers), but more likely, honest mistakes
  - Ensure class invariants survive any inputs, to minimize mutability

Copy and `clone()`

- **Mutable types often have a copy constructor that allows you to make a new instance that duplicates the value of an existing instance.**
  - In this case, `Date`'s copy constructor uses the timestamp value, measured in milliseconds since January 1, 1970.
  - As another example, `StringBuilder`'s copy constructor takes a `String`.

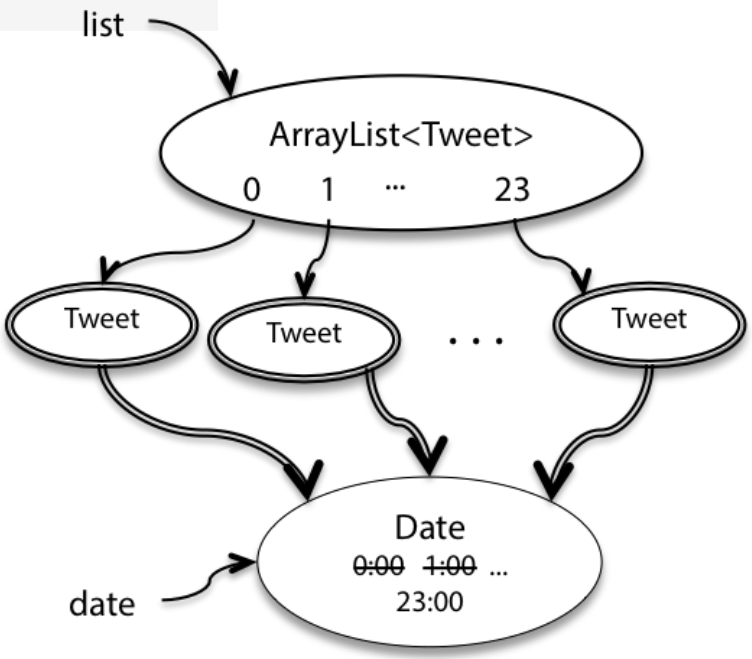
- **Another way to copy a mutable object is `clone()`, which is supported by some types but not all.**
  - To be further discussed in Chapter 8 Performance

Still rep exposure…

- **What's the side-effect of this code?**

```
/** @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("rbmllr", "keep it up! you can do it", date));
    }
    return list;
}
```

- The constructor of `Tweet` saves the reference that was passed in, so all 24 `Tweet` objects end up with the same time.



How to solve it? --- Again, defensive copying

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

- In general, you should carefully inspect the argument types and return types of all your ADT operations.
- If any of the types are mutable, make sure your implementation doesn't return direct references to its representation.
- Doing that creates rep exposure.

Leave the responsibility to your clients?

- You may object that this seems wasteful. Why make all these copies of dates? Why can't we just solve this problem by a carefully written specification, like this?

```
/**
 * 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) {
```

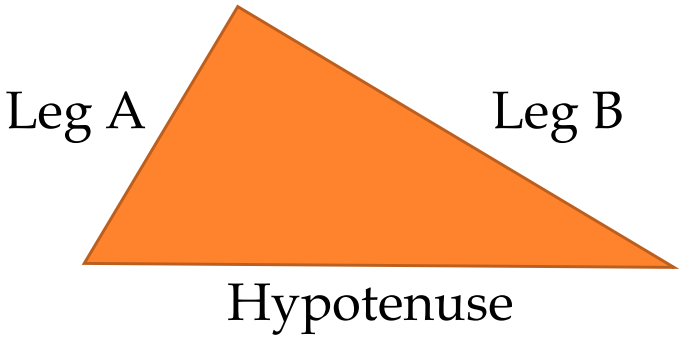
  - This approach is sometimes taken when there isn't any other reasonable alternative – for example, when the mutable object is too large to copy efficiently. 当复制代价很高时，不得不这么做
  - But the cost in your ability to reason about the program, and your ability to avoid bugs, is enormous. 但是由此引发的潜在bug也将很多

Using immutable instead of mutable types

- **In the absence of compelling arguments to the contrary, it's almost always worth it for an abstract data type to guarantee its own invariants, and preventing rep exposure is essential to that.**
  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invariants，并避免“表示泄露”。

- **An even better solution is to prefer immutable types. If we had used an immutable date object, like `java.time.ZonedDateTime`, instead of the mutable `java.util.Date`, then we would have ended this section after talking about public and private. No further rep exposure would have been possible. 最好的办法就是使用immutable的类型，彻底避免表示泄露**

An example




```java
/** Represents an immutable right triangle. */
class RightTriangle {
    private double[] sides;

    // sides[0] and sides[1] are the two legs,
    // and sides[2] is the hypotenuse, so declare it to avoid having a
    // magic number in the code:
    public static final int HYPOTENUSE = 2;

    /** Make a right triangle.
     * @param legA, legB  the two legs of the triangle
     * @param hypotenuse    the hypotenuse of the triangle.
     *        Requires hypotenuse^2 = legA^2 + legB^2
     *           (within the error tolerance of double arithmetic)
     */
    public RightTriangle(double legA, double legB, double hypotenuse) {
        this.sides = new double[] { legA, legB, hypotenuse };
    }

    /** Get all the sides of the triangle.
     *  @return three-element array with the triangle's side lengths
     */
    public double[] getAllSides() {
        return sides;
    }

    /** @return length of the triangle'
    public double getHypotenuse() {
        return sides[HYPOTENUSE];
    }
```

Representation

Constructor

Observer

```java
    /** @param factor to multiply the sides by
     *  @return a triangle made from this triangle by
     *  multiplying all side lengths by factor.
     */
    public RightTriangle scale(double factor) {
        return new RightTriangle(sides[0]*factor, sides[1]*factor, sides[2]*factor);
    }

    /** @return a regular triangle made from this triangle.
     *  A regular right triangle is one in which
     *  both legs have the same length.
     */
    public RightTriangle regularize() {
        double bigLeg = Math.max(side[0], side[1]);
        return new RightTriangle (bigLeg, bigLeg, side[2]);
    }
```

Producer

Summary

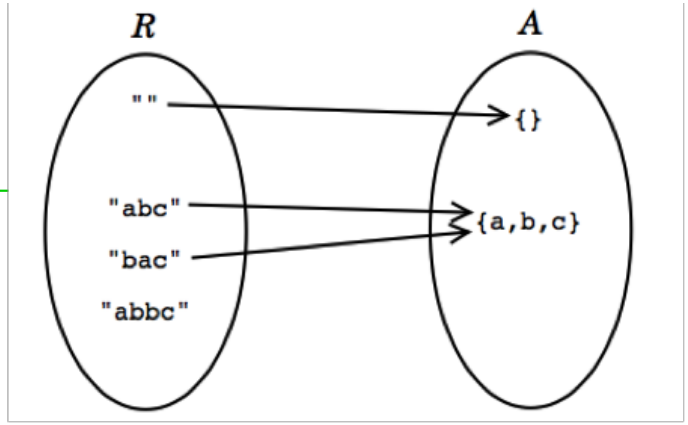
- **Don't incorporate mutable parameters into object; make defensive copies**
- **Return defensive copies of mutable fields…**
  - Return new instance instead of modifying
- **Or return *unmodifiable view* of mutable fields**
- **Real lesson – use immutable components, to eliminate the need for defensive copying**

保持不变性和避免表示泄漏，是ADT最重要的一个Invariant！

8 Rep Invariant and Abstraction Function

Theory of ADT

Two spaces of values

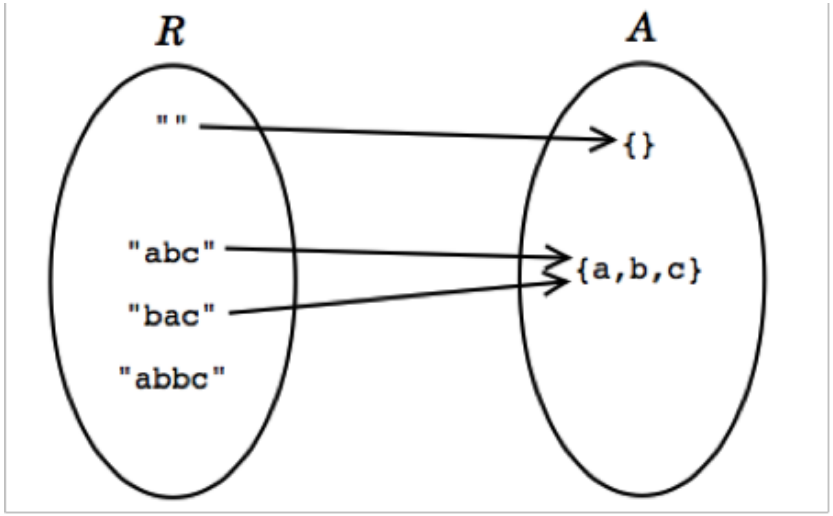


- **R: the space of representation values (rep values)** consists of the values of the actual implementation entities.
  - An ADT will be implemented as a single object, but more commonly a small network of objects is needed, so the value is often something rather complicated. 一般情况下ADT的表示比较简单，有些时候需要复杂表示
- **A: the space of abstract values** consists of the values that the type is designed to support. 抽象值构成的空间：client看到和使用的值
  - They're platonic entities that don't exist as described, but they are the way we want to view the elements of the abstract type, as clients of the type.
  - For example, an abstract type for unbounded integers might have the mathematical integers as its abstract value space; the fact that it might be implemented as an array of primitive (bounded) integers, say, is not relevant to the user of the type.

Example of two spaces

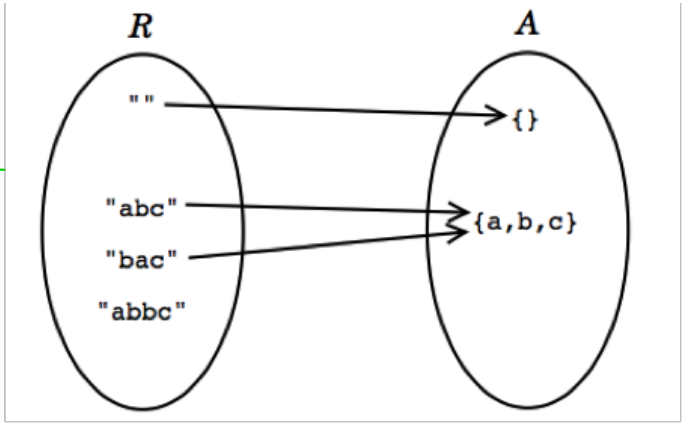
- The implementor of the abstract type must be interested in the representation values, since it is the implementor's job to achieve the illusion of the abstract value space using the rep value space. ADT开发者关注表示空间R，client关注抽象空间A
- Suppose, for example, that we choose to use a string to represent a set of characters:

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



- Then the rep space R contains Strings, and the abstract space A is mathematical sets of characters.

Mapping between R and A



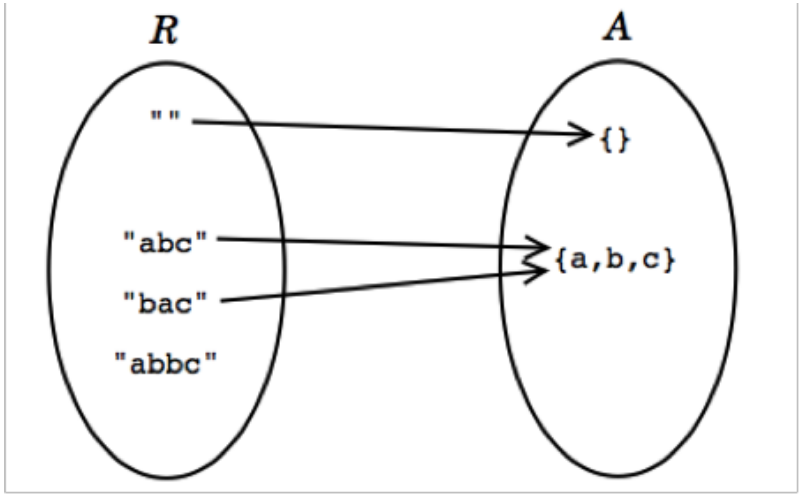
- **Every abstract value is mapped to by some rep value (surjective, 满射).**
  - The purpose of implementing the abstract type is to support operations on abstract values. Presumably, we will need to be able to create and manipulate all possible abstract values, and they must therefore be representable.
- **Some abstract values are mapped to by more than one rep value (not injective, 未必单射).**
  - This happens because the representation isn't a tight encoding. There's more than one way to represent an unordered set of characters as a string.
- **Not all rep values are mapped (not bijective, 未必双射).**
  - In this case, the string "abbc" is not mapped. In this case, we have decided that the string should not contain duplicates. This will allow us to terminate the remove method when we hit the first instance of a particular character, since we know there can be at most one.

Abstraction Function

- **An abstraction function that maps rep values to the abstract values they represent: 抽象函数：R和A之间映射关系的函数，即如何去解释R中的每一个值为A中的每一个值。**

  AF : R → A

- **The arcs in the diagram show the abstraction function.**
  - In the terminology of functions, the properties can be expressed by saying that the function is surjective (also called onto), not necessarily injective (one-to-one) and therefore not necessarily bijective, and often partial.



AF： 满射、非单射、未必双射

➔

R中的部分值并非合法的，在A中无映射值

Rep Invariant: another important ADT invariants

- **A rep invariant that maps rep values to booleans:**

```
RI : R → boolean
```

R → A

- **For a rep value `r`, `RI(r)` is true if and only if `r` is mapped by AF.**
- **In other words, RI tells us whether a given rep value is well-formed.**
- **Alternatively, you can think of RI as a set: it's the subset of rep values on which AF is defined.**

- **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**

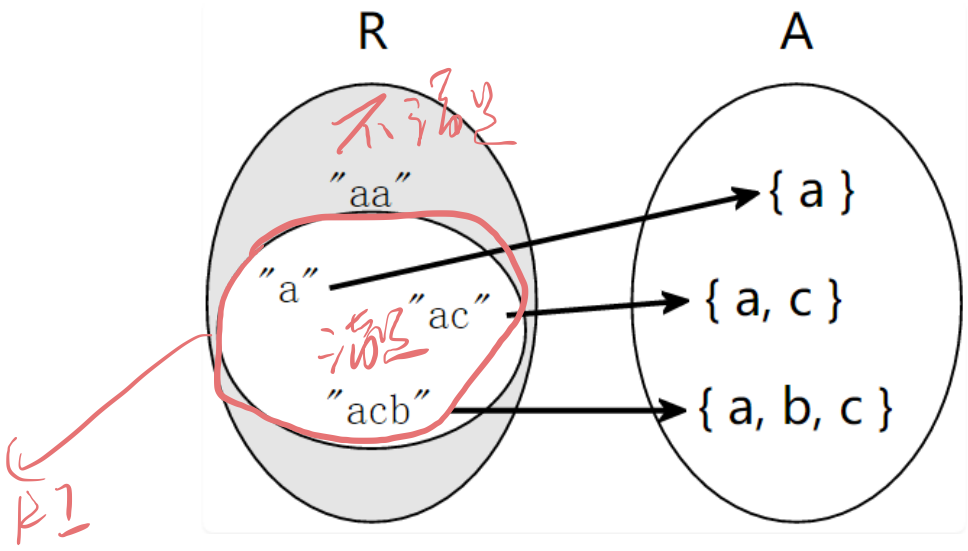
Documenting RI and AF

- **Both the rep invariant and the abstraction function should be documented in the code, right next to the declaration of the rep itself:**

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

只要不包含重复字符的字符串，都是“合法的”表示值

从s中取出所有的字符，构成集合，即为client所需的抽象值



该ADT的AF是否满足
满射？
单射？
双射？

What determine AF and RI?

- **The abstract value space alone doesn't determine AF or RI:**
  - There can be several representations for the same abstract type.
  - A set of characters could equally be represented as a string, as above, or as a bit vector, with one bit for each possible character.
  - Clearly we need two different abstraction functions to map these two different rep value spaces.
  - 不同的内部表示，需要设计不同的AF和RI

- Defining a type for the rep, and thus choosing the values for the space of rep values, does not determine which of the rep values will be deemed to be legal, and of those that are legal, how they will be interpreted. 选择某种特定的表示方式R，进而指定某个子集是“合法”的(RI)，并为该子集中的每个值做出“解释”(AF)——即如何映射到抽象空间中的值。

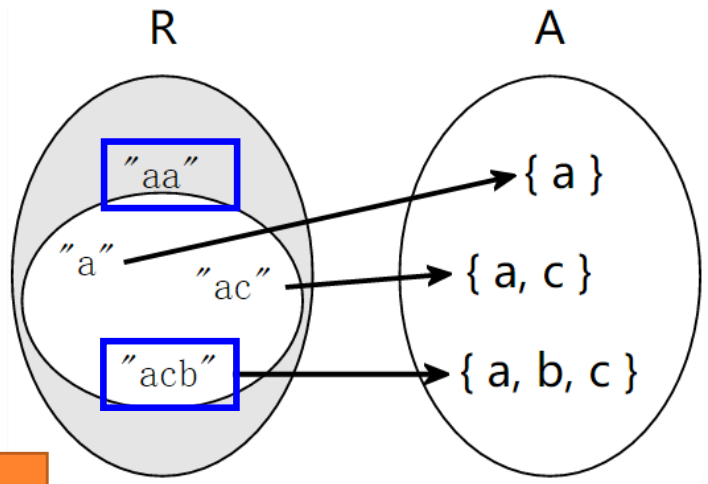
What determine AF and RI?

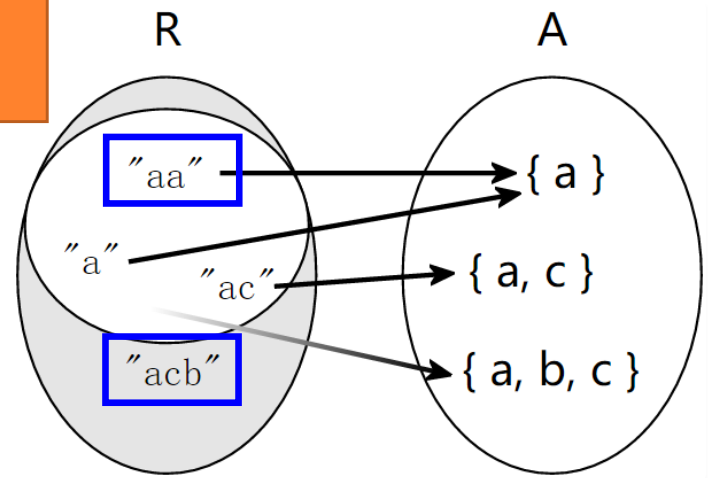
- **For example, if we allow duplicates in strings,** but at the same time require that the characters be sorted, appearing in nondecreasing order, then there would be the same rep value space but different rep invariant.

```
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[0] <= s[1] <= ... <= s[s.length()-1]
    // Abstraction Function:
    //   represents the set of characters found in s
    ...
}
```

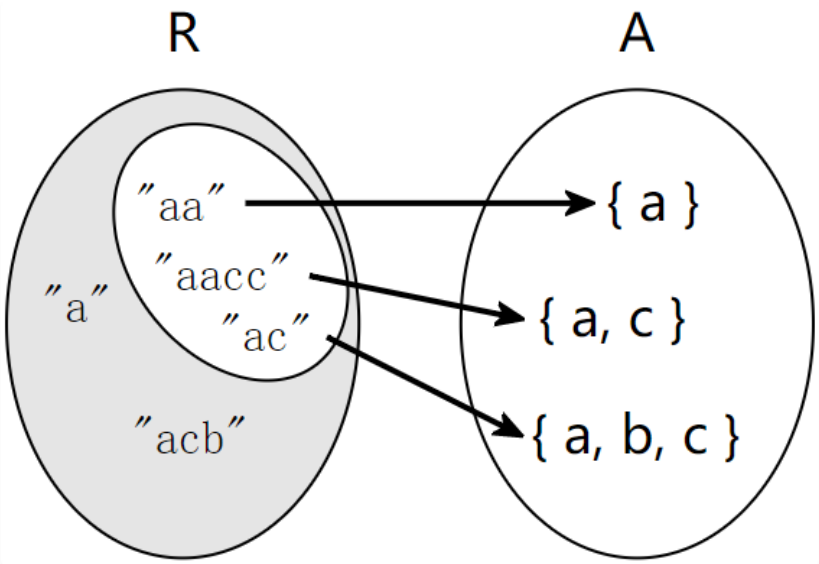
Same rep value space
Different rep invariant





What determine AF and RI?

- **Even with the same type for the rep value space and the same rep invariant RI, we might still interpret the rep differently, with different abstraction functions AF.** **即使是同样的R、同样的RI，也可能有不同的AF，即“解释不同”。**
  - Perhaps we'll interpret consecutive pairs of characters as subranges, so that the string rep `"acgg"` is interpreted as two range pairs, `[a-c]` and `[g-g]`, and therefore represents the set `{a,b,c,g}`.




```
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
    ...
}
```

Problems

- **Which of the following values of `s` satisfy this rep invariant?**
  - `"abc"`
  - `"abcd"`
  - `"eeee"`
  - `"ad"`
  - `"adad"`
  - `""`
- **Which of these values does the abstraction function map to the same abstract value as it maps `"tv"`?**
  - `"ttv"`
  - `"ttuuvv"`
  - `"ttuv"`
  - `"tuv"`

- **"Which of the following does AF(`"acfg"`) map to?**
  - `{a,b,c,d,e,f,g}`
  - `{a,b,c,f,g}`
  - `{a,c,f,g}`
  - some other abstract value
  - no abstract value, because `"acfg"` does not satisfy the rep invariant

```
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
    ...
}
```

How RI and AF influence ADT design

- The essential point is that designing an abstract type means **not only choosing the two spaces** – the abstract value space for the specification and the rep value space for the implementation – **but also deciding what rep values to use and how to interpret them**.

设计ADT：(1) 选择R和A；(2) RI --- 合法的表示值；
(3) 如何解释合法的表示值 ---映射AF

做出具体的解释：每个rep value如何映射到abstract value

- It's critically important to write down these assumptions in your code, so that future programmers (and your future self) are aware of what the representation actually means.
  - Why? What happens if different implementers disagree about the meaning of the rep?

而且要把这种选择和解释明确写到代码当中

Question 1

- **Which of the following should be known (visible and documented) to the client of an abstract data type?**
  - Abstract value space
  - Abstraction function
  - Creators
  - Observers
  - Rep
  - Rep invariant

AF是一个映射
以注释的形式

- **Which of above should be known to the developer of an abstract data type?**

Question 2

- **Suppose `C` is an abstract data type whose representation has two `String` fields:**

```
class C {
    private String s;
    private String t;
    ...
}
```

- **Assuming you don't know anything about `C`'s abstraction, which of the following might be statements in a rep invariant for `C`?**
  - `s` contains only letters
  - `s.length() == t.length()`
  - `s` represents a set of characters
  - `C`'s observers
  - `s` is the reverse of `t`
  - `s+t`

Problem 3

```
class C {
    private String s;
    ...
}
```

这两个方法分别支持哪个（些）AF/RI?

意即：执行该方法之后，RI是否仍然可以保持？

```
public void add(char c) {
    s = s + c;
}
```

Spec: 将c加入字符集合

SortedRep:

```
// AF(s) = {s[i] | 0 <= i < s.length()}
// RI: s[0] < s[1] < ... < s[s.length()-1]
```

SortedRangeRep:

```
// AF(s) = union of {s[2i],...,s[2i+1]} for 0 <= i < s.length()/2
// RI: s.length() is even, and s[0] <= s[1] <= ... <= s[s.length()-1]
```

NoRepeatsRep:

```
// AF(s) = {s[i] | 0 <= i < s.length()}
// RI: s contains no character more than once
```

AnyRep:

```
// AF(s) = {s[i] | 0 <= i < s.length()}
// RI: true
```

无RI限制

```
public void remove(char c) {
    int position = s.indexOf(c);
    if (position >= 0) {
        s = s.substring(0, position) +
            s.substring(position+1, s.length());
    }
}
```

xyyx→yyx

Spec: 将c从字符集合中删除

Problem 3

```
class C {
    private String s;
    ...
}
```

这个方法分别支持哪个（些）AF/RI?

SortedRep:

```
// AF(s) = {s[i] | 0 <= i < s.length()}
// RI: s[0] < s[1] < ... < s[s.length()-1]
```

SortedRangeRep:

```
// AF(s) = union of {s[2i],...,s[2i+1]} for 0 <= i < s.length()/2
// RI: s.length() is even, and s[0] <= s[1] <= ... <= s[s.length()-1]
```

NoRepeatsRep:

```
// AF(s) = {s[i] | 0 <= i < s.length()}
// RI: s contains no character more than once
```

AnyRep:

```
// AF(s) = {s[i] | 0 <= i < s.length()}
// RI: true
```

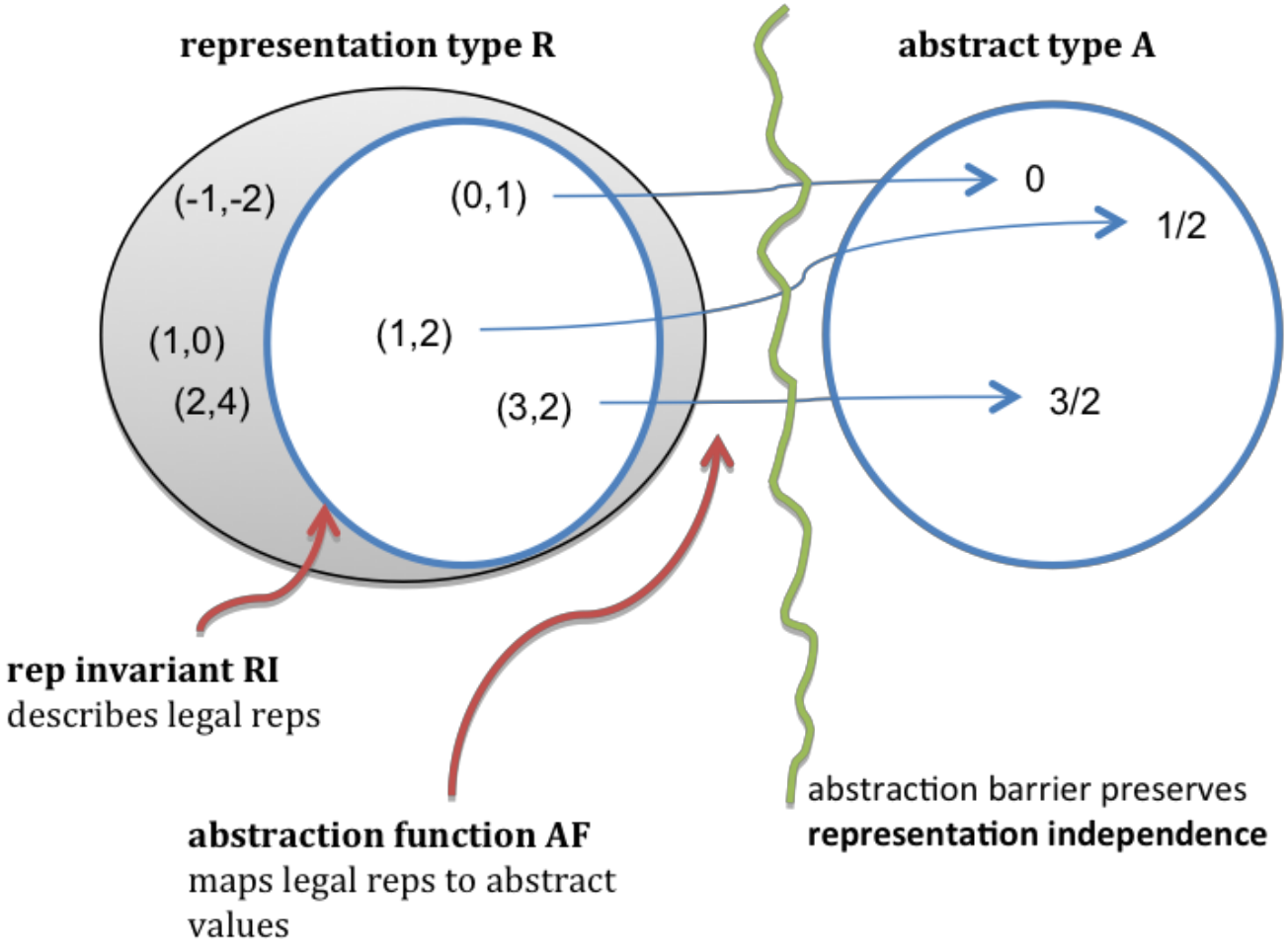
```
public boolean contains(char c) {
    for (int i = 0; i < s.length(); i += 2) {
        char low = s.charAt(i);
        char high = s.charAt(i+1);
        if (low <= c && c <= high) {return true;}
    }
    return false;
}
```

Example: ADT for Rational Numbers

```
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

    /** Make a new Ratnum == n.
     *  @param n value */
    public RatNum(int n) {
        numer = n;
        denom = 1;
        checkRep();
    }

    /** Make a new Ratnum == n.
     *  @param n value */
    public RatNum(int n) {
        numer = n;
        denom = 1;
        checkRep();
    }

    /** Make a new RatNum == (n / d).
     *  @param n numerator
     *  @param d denominator
     *  @throws ArithmeticException if d == 0 */
    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();
    }
}
```

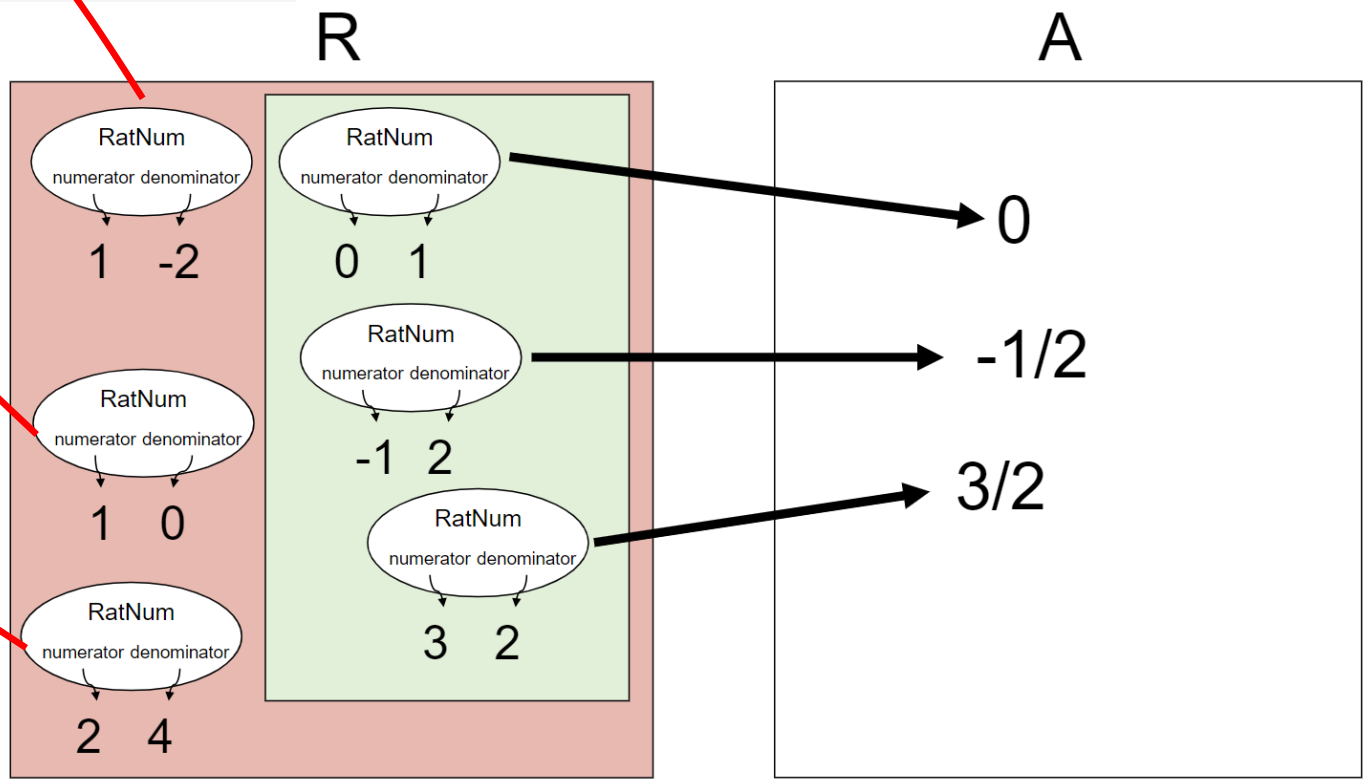
RI and AF of this example

- The RI requires that numerator/denominator pairs be in reduced form (i.e., lowest terms), so pairs like (2,4) and (18,12) above should be drawn as outside the RI.



RI and AF of this example

```
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

    /** Make a new Ratnum == n.
     *  @param n value */
    public RatNum(int n) {
        numer = n;
        denom = 1;
        checkRep();
    }
```



Checking the Rep Invariant 随时检查RI是否满足

- The rep invariant isn't just a neat mathematical idea. If your implementation asserts the rep invariant at run time, then you can catch bugs early.

```
// Check that the rep invariant is true
// *** Warning: this does nothing unless you turn on assertion checking
// by passing -enableassertions to Java
private void checkRep() {
    assert denom > 0;
    assert gcd(numer, denom) == 1;
}
```

- You should certainly call `checkRep()` to assert the rep invariant at the end of every operation that creates or mutates the rep (creators, producers, and mutators). 在所有可能改变rep的方法内都要检查
- Observer methods don't normally need to call `checkRep()`, but it's good defensive practice to do so anyway. Observer方法可以不用，但建议也要检查，以防止你的“万一”
  - Calling `checkRep()` in every method, including observers, means you'll be more likely to catch rep invariant violations caused by rep exposure.

9 Beneficent mutation

Beneficent mutation

- Recall that a type is immutable if and only if a value of the type never changes after being created.
- With our new understanding of the abstract space A and rep space R, we can refine this definition: the *abstract value* should never change.
- But the implementation is free to mutate a *rep value* as long as it continues to map to the same abstract value, so that the change is invisible to the client.
- This kind of change is called **beneficent mutation** (有益的可变性).

- 对immutable的ADT来说，它在A空间的abstract value应是不变的。
- 但其内部表示的R空间中的取值则可以是变化的。

An example of beneficent mutation

- `RatNum`: this rep has a weaker rep invariant that doesn't require the numerator and denominator to be stored in lowest terms

```
public class RatNum {

    private int numerator;
    private int denominator;

    // Rep invariant:
    //   denominator != 0

    // Abstraction function:
    //   AF(numerator, denominator) = numerator/denominator

    /**
     * Make a new RatNum == (n / d).
     * @param n numerator
     * @param d denominator
     * @throws ArithmeticException if d == 0
     */
    public RatNum(int n, int d) throws ArithmeticException {
        if (d == 0) throw new ArithmeticException();
        numerator = n;
        denominator = d;
        checkRep();
    }

    ...
}
```

An example of beneficent mutation

- **This weaker rep invariant allows a sequence of `RatNum` arithmetic operations to simply omit reducing the result to lowest terms. But when it's time to display a result to a human, we first simplify it:**

```
/**
 * @return a string representation of this rational number
 */
@Override
public String toString() {
    int g = gcd(numerator, denominator);
    numerator /= g;
    denominator /= g;
    if (denominator < 0) {
        numerator = -numerator;
        denominator = -denominator;
    }
    checkRep();
    return (denominator > 1) ? (numerator + "/" + denominator)
                             : (numerator + "");
}
```

An example of beneficent mutation

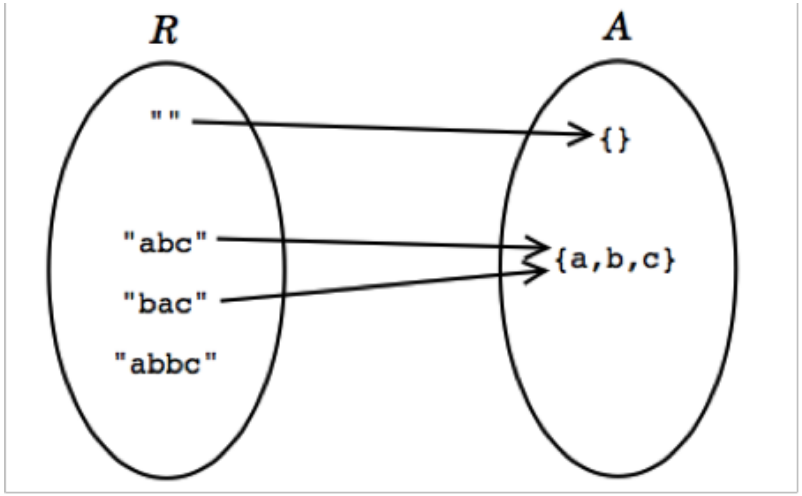
- **Notice that this `toString` implementation reassigns the private fields numerator and denominator, mutating the representation – even though it is an observer method on an immutable type!**
- **But, crucially, the mutation doesn't change the abstract value.**
  - Dividing both numerator and denominator by the same common factor, or multiplying both by -1, has no effect on the result of the abstraction function, AF(numerator, denominator) = numerator/denominator.
- **Another way of thinking about it: AF is a many-to-one function, and the rep value has changed to another that still maps to the same abstract value. So the mutation is harmless, or beneficent.**

- **这种mutation只是改变了R值，并未改变A值，对client来说是immutable的 → "AF并非单射"，从一个R值变成了另一个R值**
- **但这并不代表在immutable的类中就可以随意出现mutator！**

Why is beneficent mutation required?

- **This kind of implementer freedom often permits performance improvements like:**
  - Caching：例如通过cache暂存某些频繁计算的结果
  - Data structure rebalancing：例如对tree数据结构进行插入或删除节点之后
  - Lazy computation：上例中在toString()的时候才进行约分计算

- **通过牺牲immutability的部分原则来换取“效率”和“性能”**



AF： 满射、非单射、未必双射

➔ A值在R中有多个对应的R值

➔ 可以内部改变R值，不改变A值

10 Documenting the AF, RI, and Safety from Rep Exposure

Documenting AF and RI

- **It's good practice to document the abstraction function and rep invariant in the class**, using comments right where the private fields of the rep are declared. 在代码中用注释形式记录AF和RI

- It is not enough for the RI to be a generic statement like "all fields are valid." 要精确的记录RI：rep中的所有fields何为有效
  - The job of the rep invariant is to explain precisely what makes the field values valid or not.
- It is not enough for the AF to offer a generic explanation like "represents a set of characters." 要精确记录AF：如何解释每一个R值
  - The job of the abstraction function is to define precisely how the concrete field values are interpreted.
  - As a *function*, if we take the documented AF and substitute in actual (legal) field values, we should obtain out a complete description of the single abstract value they represent.

Documenting rep exposure safety argument

- Another piece of documentation is a **rep exposure safety argument** 表示泄漏的安全声明
- This is a comment that examines each part of the rep, looks at the code that handles that part of the rep (particularly with respect to parameters and return values from clients, because that is where rep exposure occurs), and presents a reason why the code doesn't expose the rep. 给出理由，证明代码并未对外泄露其内部表示——自证清白

Example 1

```
// Immutable type representing a tweet.
public class Tweet {

    private final String author;
    private final String text;
    private final Date timestamp;

    // Rep invariant:
    //   author is a Twitter username (a nonempty string of letters, digits, underscores)
    //   text.length <= 280
    // Abstraction function:
    //   AF(author, text, timestamp) = a tweet posted by author, with content text,
    //                                 at time timestamp
    // Safety from rep exposure:
    //   All fields are private;
    //   author and text are Strings, so are guaranteed immutable;
    //   timestamp is a mutable Date, so Tweet() constructor and getTimestamp()
    //       make defensive copies to avoid sharing the rep's Date object with

    // Operations (specs and method bodies omitted to save space)
    public Tweet(String author, String text, Date timestamp) { ... }
    public String getAuthor() { ... }
    public String getText() { ... }
    public Date getTimestamp() { ... }
}
```

RI：针对Rep的每一个field以及多个fields之间的关系，进行条件限定，要精确

防止Rep泄漏

AF：给出client看到的A值是什么，是对每一个Rep值的“数学运算”

Example 2

```
// Immutable type representing a rational number.
public class RatNum {
    private final int numer;
    private final int denom;

    // Rep invariant:
    //   denom > 0
    //   numer/denom is in reduced form, i.e. gcd(|numer|,denom) = 1
    // Abstraction Function:
    //   represents the rational number numer / denom
    // Safety from rep exposure:
    //   All fields are private, and all types in the rep are immutable.

    // Operations (specs and method bodies omitted to save space)
    public RatNum(int n) { ... }
    public RatNum(int n, int d) throws ArithmeticException { ... }
    ...
}
```

Example 3

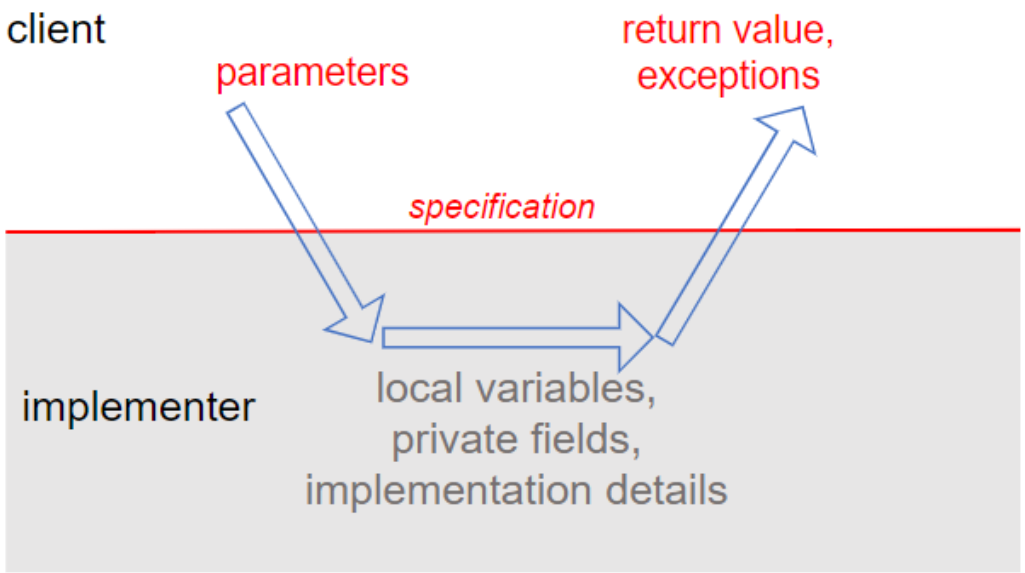
```
// Mutable type representing Twitter users' followers.
public class FollowGraph {
    private final Map<String, Set<String>> followersOf;

    // Rep invariant:
    //    all Strings in followersOf are Twitter usernames
    //           (i.e., nonempty strings of letters, digits, underscores)
    //    no user follows themselves, i.e. x is not in followersOf.get(x)
    // Abstraction function:
    //    AF(followersOf) = the follower graph where Twitter user x is followed by user y
    //                      if and only if followersOf.get(x).contains(y)
    // Safety from rep exposure:
    //    All fields are private, and ..???..

    // Operations (specs and method bodies omitted to save space)
    public FollowGraph() { ... }
    public void addFollower(String user, String follower) { ... }
    public void removeFollower(String user, String follower) { ... }
    public Set<String> getFollowers(String user) { ... }
}
```

`getFollowers()` makes a defensive copy of the `Set` it returns, and all other parameters and return values are immutable `String` or `void`.

The `Set` objects in the rep are made immutable by unmodifiable wrappers

Summary: What an ADT spec may talk about

- The specification of an abstract type *T* – which consists of the specs of its operations – should only talk about things that are visible to the client. ADT的规约里只能使用client可见的内容来撰写，包括参数、返回值、异常等。
  - This includes parameters, return values, and exceptions thrown by its operations.
- Whenever the spec needs to refer to a value of type *T*, it should describe the value as an *abstract value*, i.e. mathematical values in the abstract space A. 如果规约里需要提及“值”，只能使用A空间中的“值”。



Summary: What an ADT spec may talk about

- The spec should not talk about details of the representation, or elements of the rep space R. ADT的规约里也不应谈及任何内部表示的细节，以及R空间中的任何值
- It should consider the rep itself (the private fields) invisible to the client, just as method bodies and their local variables are considered invisible. ADT的内部表示(私有属性)对外部都应严格不可见
- That's why we write the rep invariant and abstraction function as ordinary comments within the body of the class, rather than Javadoc comments above the class. 故在代码中以注释的形式写出AF和RI而不能在Javadoc文档中，防止被外部看到而破坏表示独立性/信息隐藏
  - Writing them as Javadoc comments would commit to them as public parts of the type's specification, which would intefere with rep independence and information hiding.

How to establish invariants

- An invariant is a property that is true for the entire program – which in the case of an invariant about an object, reduces to the entire lifetime of the object.
- **To make an invariant hold, we need to:**
  - Make the invariant true in the initial state of the object;
  - Ensure that all changes to the object keep the invariant true.
  - 在对象的初始状态不变量为true，在对象发生变化时，不变量也要为true

应该在每一个创建或改变（构造器，生产器，变值器）表示数据的操作后调用checkRep()检查表示不变量

- **Translating this in terms of the types of ADT operations:**
  - Creators and producers must establish the invariant for new object instances; 构造器和生产器在创建对象时要确保不变量为true
  - Mutators and observers must preserve the invariant. 变值器和观察器在执行时必须保持不变性。
  - Using `checkRep()` to check invariants before each method returns. 在每个方法return之前，用`checkRep()`检查不变量是否得以保持。

How to establish invariants

- The risk of rep exposure makes the situation more complicated.
- If the rep is exposed, then the object might be changed anywhere in the program, not just in the ADT's operations, and we can't guarantee that the invariant still holds after those arbitrary changes.
- 表示泄漏的风险：一旦泄露，ADT内部表示可能会在程序的任何位置发生改变（而不是限制在ADT内部），从而无法确保ADT的不变量是否能够始终保持为true。
- So the full rule for proving invariants— —If an invariant of an abstract data type is
  - established by creators and producers;
  - preserved by mutators, and observers;
  - no representation exposure occurs,

  then the invariant is true of all instances of the abstract data type.

用这三个标准来检查你的ADT是否保持不变量？

Does this ADT keep its invariant?

```java
/** Represents an immutable right triangle. */
class RightTriangle {
    private double[] sides;
    // RI: ???
    // AF: ???

    // sides[0] and sides[1] are the two legs,
    // and sides[2] is the hypotenuse, so declare it to avoid having a
    // magic number in the code:
    public static final int HYPOTENUSE = 2;

    /** Make a right triangle.
     * @param legA, legB  the two legs of the triangle
     * @param hypotenuse    the hypotenuse of the triangle.
     *        Requires hypotenuse^2 = legA^2 + legB^2
     *           (within the error tolerance of double arithmetic)
     */
    public RightTriangle(double legA, double legB, double hypotenuse) {
        this.sides = new double[] { legA, legB, hypotenuse };
    }

    /** Get all the sides of the triangle.
     *  @return three-element array with the triangle's side lengths
     */
    public double[] getAllSides() {
        return sides;
    }

    /** @return length of the triangle's hypotenuse
    public double getHypotenuse() {
        return sides[HYPOTENUSE];
    }
```

Here causes a rep exposure, so the client may inadvertently change values in the returned array and destroy the invariant as a result, even while obeying all the specs as written.

This method creates a new triangle, but changes only the legs and doesn't recalculate a new hypotenuse for it. This doesn't preserve the Pythagorean invariant for the new triangle.

```java
    /** @param factor to multiply the sides by
     *  @return a triangle made from this triangle by
     *  multiplying all side lengths by factor.
     */
    public RightTriangle scale(double factor) {
        return new RightTriangle (sides[0]*factor, sides[1]*factor, sides[2]*factor);
    }

    /** @return a regular triangle made from this triangle.
     *  A regular right triangle is one in which
     *  both legs have the same length.
     */
    public RightTriangle regularize() {
        double bigLeg = Math.max(sides[0], sides[1]);
        return new RightTriangle (bigLeg, bigLeg, sides[2]);
    }
}
```

往届学生对ADT学习的感悟

系列小结

好啦，我们现在看一下整个的流程。我们首先达成了一个和client的契约与合同：spec，其中限制了client的输入和implement的返回输出，其中涉及了很多抽象的元素；之后，我们将这个抽象的东西一步步落实，作为程序员，在implement的过程中，我们使用R->RI->AF->A的流程将抽象的数据和我们现有的数据类型映射建立联系，表达出来。

ADT核心在于其结构上的方法，也就是所谓的运算，以所具有的方法来理解ADT是一个良好的理解方式。从抽象角度来看，所有这些无非就是抽象集群之间的变化。而我们做的，其实就是在不让外界了解到任何内部的结构，不让外界获取或者修改内部任何信息的条件下，实现了现有结构和抽象集群及其方法的一一对应，将其表达。

类比理解，笔者的感慨

spec到最后实现的ADT，是用抽象实现抽象的过程，他们的关系既像那些家庭里面扛下一些重担，给孩子外表光鲜的面容的父亲做下的许诺；也像极了爱情中，一方对另一方的求全责备，难以设身处地，却仍然得到的满足。
尽情许诺吧。每个人都是一个某个人的造梦者每个人也都是某个人的圆梦人。

可能从我们的祖先在山洞里将钻木的火苗留下，用草垛围起不让他熄灭的的那一刻起，世界就开始了他的层层封装吧。从跳跃的电子到迸射火光的希望，从摩擦的齿轮到隆隆的机械的争抢，从轻轻转动的纤细磁臂到高压作业的电压计的稀松平常，从单晶硅片的电路到超级算力下带来的信息充斥的迷惘。

我们于是也将以前的自己封印在了过去的躯壳，我们带着将过去包装的万无一失的自己，再看不到曾经的点滴。

不相信眼泪，毕竟，rep对于除了你以外的人，没有意义，（想苦诉衷肠却只会rep exposure）。

总之祝好，希望您也是field里面有属性的ADT

11 ADT invariants replace preconditions

ADT invariants replace preconditions

- An enormous advantage of a well-designed abstract data type is that it encapsulates and enforces properties that we would otherwise have to stipulate in a precondition. 用ADT不变量取代复杂的Precondition，相当于将复杂的precondition封装到了ADT内部。

```
/**
 * @param set1 is a sorted set of characters with no repeats
 * @param set2 is likewise
 * @return characters that appear in one set but not the other,
 *  in sorted order with no repeats
 */
static String exclusiveOr(String set1, String set2);
```

```
/** @return characters that appear in one set but not the other */
static SortedSet<Character> exclusiveOr(SortedSet<Character>  set1, SortedSet<Character
> set2);
```

ADT invariants replace preconditions

- **It's safer from bugs, because the required condition (sorted with no repeats) can be enforced in exactly one place, the `SortedSet` type, and because Java static checking comes into play, preventing values that don't satisfy this condition from being used at all, with an error at compile-time.**

- **It's easier to understand, because it's much simpler, and the name `SortedSet` conveys what the programmer needs to know.**

- **It's more ready for change, because the representation of `SortedSet` can now be changed without changing `exclusiveOr` or any of its clients.**

An Example

```
/**
 * Find tweets written by a particular user.
 *
 * @param tweets a list of tweets with distinct timestamps, not modified by this method.
 * @param username Twitter username (a nonempty sequence of letters, digits, and underscore)
 * @return all and only the tweets in the list whose author is username,
 *         in the same order as in the input list.
 */
public static List<Tweet> writtenBy(List<Tweet> tweets, String username) { ... }
```

```
public static List<Tweet>
                    writtenBy(TweetList tweets, UserName username)
{ ... }
```

`TweetList` would be able to represent the requirement that the tweets have distinct timestamps

`UserName` would be able to represent the constraint on valid usernames.

Summary

Summary

- **Abstract data types are characterized by their operations.**
- **Operations can be classified into creators, producers, observers, and mutators.**
- **An ADT's specification is its set of operations and their specs.**
- **A good ADT is simple, coherent, adequate, and representation-independent.**
- **An ADT is tested by generating tests for each of its operations, but using the creators, producers, mutators, and observers together in the same tests.**

Summary

- **Safe from bugs.** A good ADT offers a well-defined contract for a data type, so that clients know what to expect from the data type, and implementors have well-defined freedom to vary.
- **Easy to understand.** A good ADT hides its implementation behind a set of simple operations, so that programmers using the ADT only need to understand the operations, not the details of the implementation.
- **Ready for change.** Representation independence allows the implementation of an abstract data type to change without requiring changes from its clients.

Summary

- **An invariant is a property that is always true of an ADT object instance, for the lifetime of the object.**
- **A good ADT preserves its own invariants. Invariants must be established by creators and producers, and preserved by observers and mutators.**
- **The rep invariant specifies legal values of the representation, and should be checked at runtime with `checkRep()`.**
- **The abstraction function maps a concrete representation to the abstract value it represents.**
- **Representation exposure threatens both representation independence and invariant preservation.**

Summary

- **Safe from bugs.** A good ADT preserves its own invariants, so that those invariants are less vulnerable to bugs in the ADT's clients, and violations of the invariants can be more easily isolated within the implementation of the ADT itself. Stating the rep invariant explicitly, and checking it at runtime with checkRep(), catches misunderstandings and bugs earlier, rather than continuing on with a corrupt data structure.
- **Easy to understand.** Rep invariants and abstraction functions explicate the meaning of a data type's representation, and how it relates to its abstraction.
- **Ready for change.** Abstract data types separate the abstraction from the concrete representation, which makes it possible to change the representation without having to change client code.

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

The end

March 24, 2024