




Chapter 3: Abstract Data Type (ADT) and Object-Oriented Programming (OOP)

3.3 Abstract Data Type (ADT)


Xu Hanchuan
xhc@hit.edu.cn

March 20, 2018

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**

Outline

- 
1. Invariants
 2. Rep Invariant and Abstraction Function
 3. Beneficent mutation
 4. Documenting the AF, RI, and Safety from Rep Exposure
 5. ADT invariants replace preconditions
 6. Summary

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.

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.



1 Abstraction and User-Defined Types




What *abstraction* means

- **Abstract data types are an instance of a general principle in software engineering, which goes by many names:**
 - **Abstraction.** Omitting or hiding low-level details with a simpler, higher-level idea.
 - **Modularity.** Dividing a system into components or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system.
 - **Encapsulation.** Building walls around a module (a hard shell or capsule) so that the module is responsible for its own internal behavior, and bugs in other parts of the system can't damage its integrity.
 - **Information hiding.** Hiding details of a module's implementation from the rest of the system, so that those details can be changed later without changing the rest of the system.
 - **Separation of concerns.** Making a feature (or “concern”) the responsibility of a single module, rather than spreading it across multiple modules.

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**.

SE researchers who made contributions

- 
- **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)
 - **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)

Data Abstraction

- **Data abstraction: a type is characterized by the operations you can perform on it.**
 - A number is something you can add and multiply;
 - A string is something you can concatenate and take substrings of;
 - A boolean is something you can negate, and so on.
- In a sense, users could already define their own types in early programming languages: you could create a record type `date`, for example, with `integer` fields for `day`, `month`, and `year`.
- But what made abstract types new and different was the focus on **operations**: the user of the type would not need to worry about how its values were actually stored, in the same way that a programmer can ignore how the compiler actually stores integers. All that matters is the **operations**.



2 Classifying Types and Operations



Mutable and immutable types

- **Types, whether built-in or user-defined, can be classified as mutable or immutable .**
 - 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.
 - So `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.
 - Sometimes a type will be provided in two forms, a mutable and an immutable form. `StringBuilder`, for example, is a mutable version of `String` (although the 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.
 - 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.
 - 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.
 - The `size()` method of `List`, for example, returns an `int`.
- **Mutators** change objects.
 - 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** : $t^* \rightarrow T$
- **producer** : $T+, t^* \rightarrow T$
- **observer** : $T+, t^* \rightarrow t$
- **mutator** : $T+, t^* \rightarrow \text{void} \mid t \mid T$

- 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 \mid 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`

Signature of a creator

- A creator is either implemented as a constructor , like `new ArrayList()`, or simply a static method instead, like `Arrays.asList()`.
 - A creator implemented as a static method is often called a **factory method**.
 - The various `String.valueOf()` methods in Java are other examples of creators implemented as factory methods.

Signature of a mutator

- Mutators are often signaled by a `void` return type.
- A method that returns `void` must be called for some kind of side-effect, since otherwise it doesn't return anything.
- But not all mutators return `void`.
 - 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

- **int is Java's primitive integer type. 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)

List

- **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. These classes include ArrayList and LinkedList .**
 - creators: ArrayList and LinkedList constructors, Collections.singletonList
 - producers: Collections.unmodifiableList
 - observers: size , get
 - mutators: add , remove , addAll , Collections.sort

String

- **String is Java's string type. String is immutable.**
 - creators: String constructors
 - producers: concat , substring , toUpperCase
 - observers: length , charAt
 - mutators: none (it's immutable)

Realizing ADT Concepts in Java

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



4 Designing an Abstract Type



Designing an Abstract Type

- Designing an abstract type involves choosing good operations and determining how they should behave.
- **Rules of thumb 1**
 - 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
- **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.
 - 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.

Designing an Abstract Type

- Rules of thumb 3
- **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 .



5 Representation Independence



Representation Independence

- Critically, a good abstract data type should be **representation independent**.
 - This means that 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.

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) { ... }
}
```

A simple representation for MyString

- For now, let's 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;
}
```

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;
}
```

What is Representation Independence


- 
- 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.



6 Testing an Abstract Data Type



How to test an ADT

- 
- We build a test suite for an abstract data type 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.

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()
```

Test suite that covers all partitions

```
@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



Invariants of an ADT

- The most important property of a good abstract data type is that it **preserves its own invariants**.
- An *invariant* is a property of a program that is always true, for every possible runtime state of the program.
 - Immutability is one crucial invariant: once created, an immutable object should always represent the same value, for its entire lifetime.
- Saying that the **ADT** *preserves its own invariants* means that the ADT is responsible for ensuring that its own invariants hold.
 - It doesn't depend on good behavior from its clients.
 - Correctness doesn't depend on other modules.
- **Invariant is established by the constructor**
 - Maintained by public method invocations
 - May be invalidated temporarily during method execution

Why are invariants required?

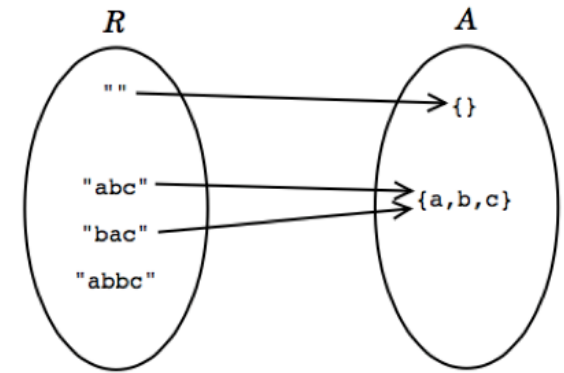
- When an ADT preserves its own invariants, reasoning about the code becomes much easier.
- If you can count on the fact that Strings never change, you can rule out that possibility when you're debugging code that uses Strings – or when you're trying to establish an invariant for another ADT that uses 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.
- ... **Assume clients will try to destroy invariants (malicious hackers or honest mistakes)**



8 Rep Invariant and Abstraction Function



Two spaces of values

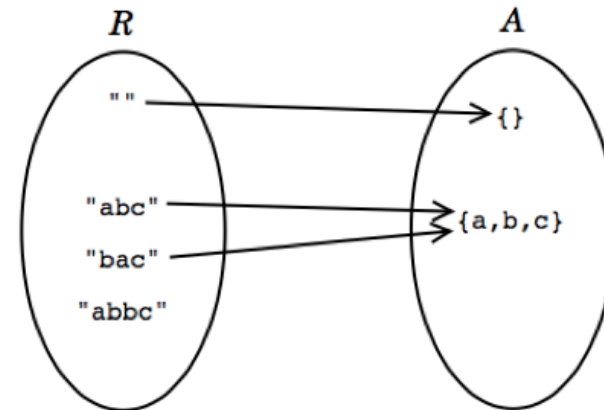


- **R: the space of representation values (rep values)** consists of the values of the actual implementation entities.
 - In simple cases, an abstract type will be implemented as a single object, but more commonly a small network of objects is needed, so this value is actually often something rather complicated.
- **A: the space of abstract values** consists of the values that the type is designed to support.
 - 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

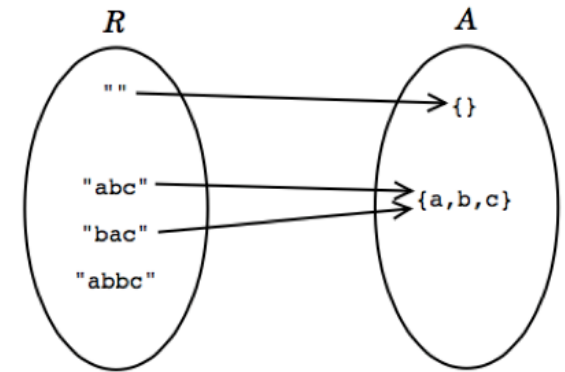
- The implementor of the abstract type must be interested in the representation values, since it is the implementor's job to achieve the illusion of the abstract value space using the rep value space.
- Suppose, for example, that we choose to use a string to represent a set of characters:

```
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



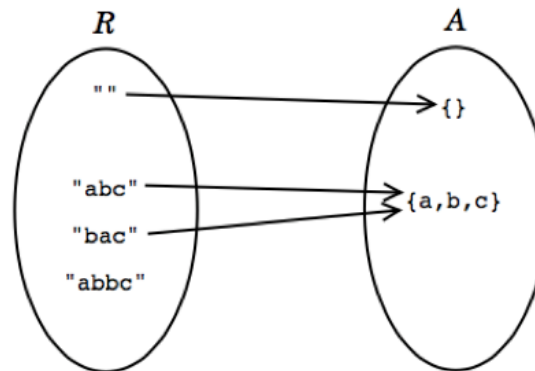
- **Every abstract value is mapped to by some rep value (surjective, 满射).**
 - The purpose of implementing the abstract type is to support operations on abstract values. Presumably, then, we will need to be able to create and manipulate all possible abstract values, and they must therefore be representable.
- **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:

$$AF : R \rightarrow A$$

- 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.



Rep Invariant

- A rep invariant that maps rep values to booleans:

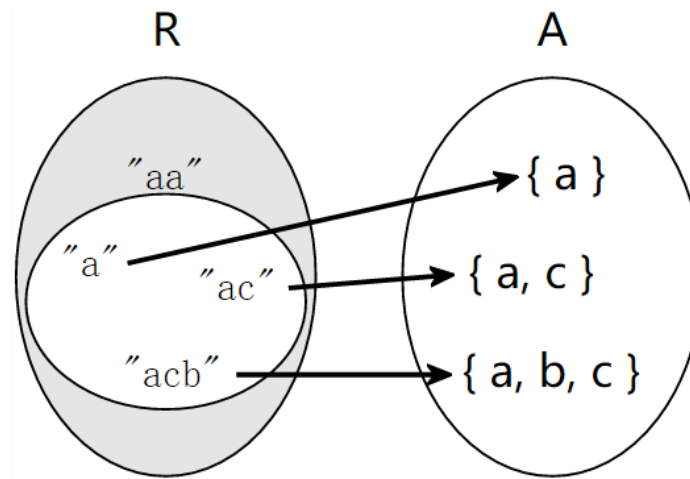
$RI : R \rightarrow \text{boolean}$

- 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.

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:  
    //   represents the set of characters found in s  
    ...  
}
```



What determine AF and RI?

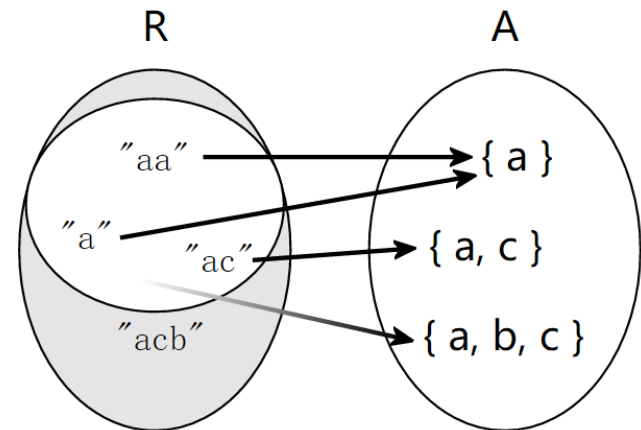
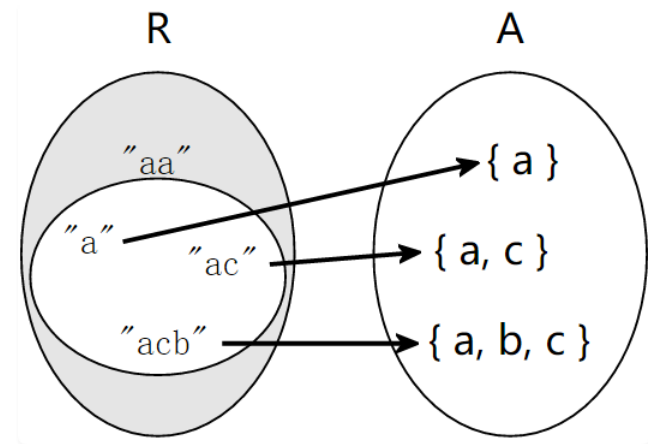
- **The abstract value space alone doesn't determine AF or RI:**
 - There can be several representations for the same abstract type.
 - A set of characters could equally be represented as a string, as above, or as a bit vector, with one bit for each possible character. Clearly we need two different abstraction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

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
    ...
}
```



What determine AF and RI?

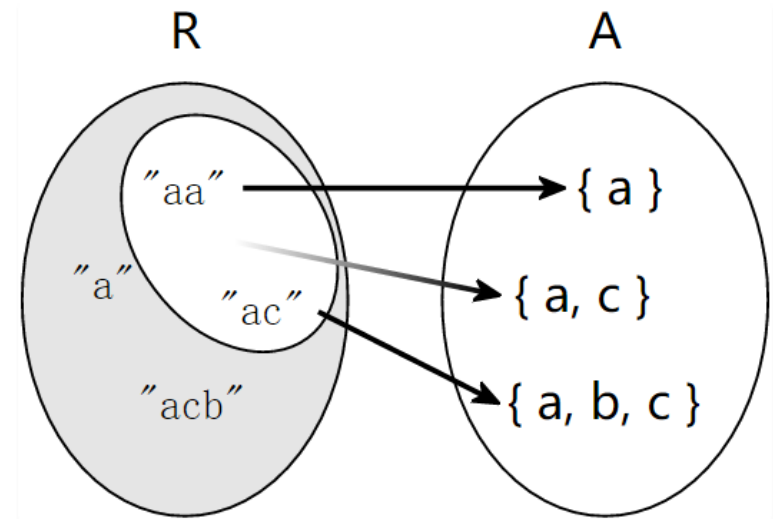
- Even with the same type for the rep value space and the same rep invariant RI, we might still interpret the rep differently, with different abstraction functions AF.
- Suppose RI admits any string of characters. Then we could define AF, as above, to interpret the array's elements as the elements of the set. But there's no a priori reason to let the rep decide the interpretation.
- 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}.
- Here's what the AF and RI would look like for that representation.

What determine AF and RI?

```

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
    ...
}

```



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** .
- It's critically important to write down these assumptions in your code, as we've done above, 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?

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();
    }
}
```

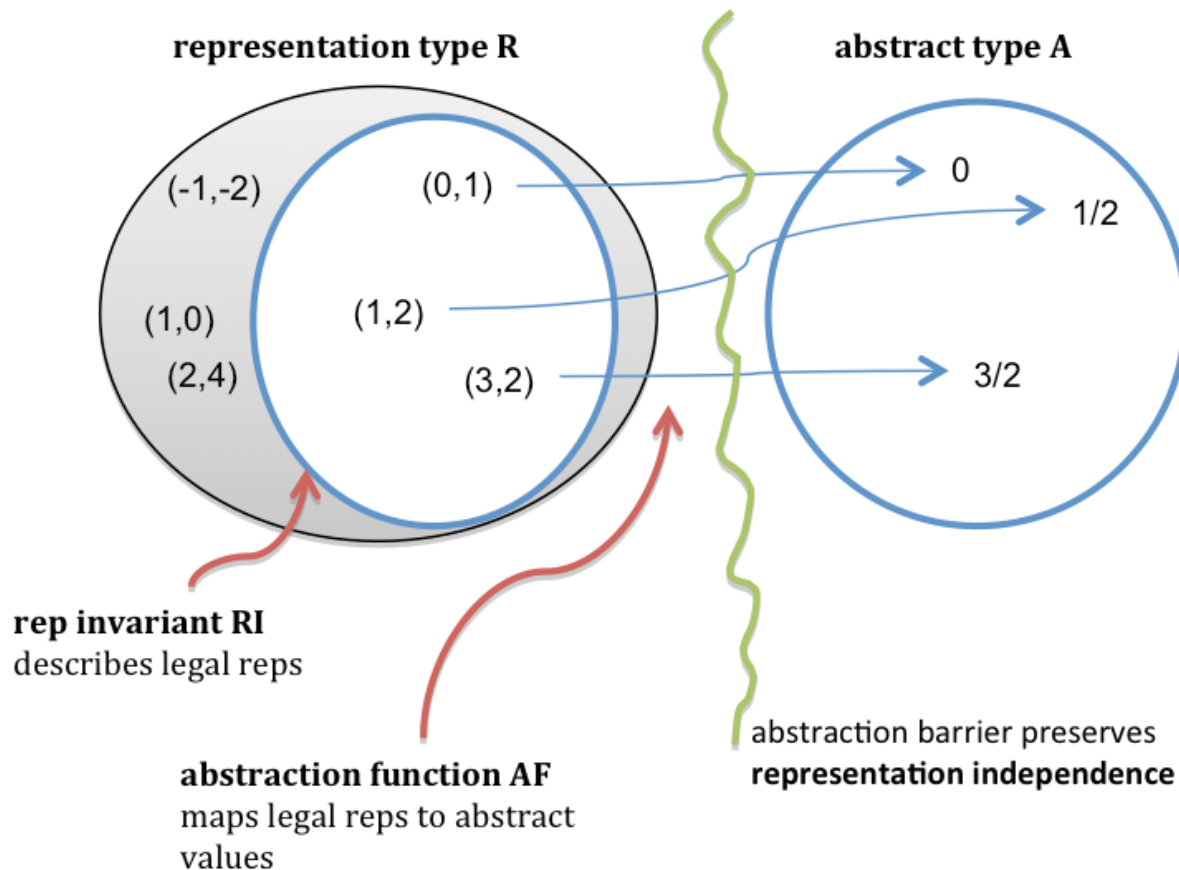
```
/** Make a new Ratnum == n.
 * @param n value */
public RatNum(int n) {
    numer = n;
    denom = 1;
    checkRep();
}

/** 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.



Checking the Rep Invariant

- 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). Look back at the `RatNum` code above, and you'll see that it calls `checkRep()` at the end of both constructors.
- Observer methods don't normally need to call `checkRep()`, but it's good defensive practice to do so anyway.
 - Calling `checkRep()` in every method, including observers, means you'll be more likely to catch rep invariant violations caused by rep exposure.

No Null Values in the Rep

- null values are troublesome and unsafe, so that we try to remove them from our programming entirely.
- The preconditions and postconditions of our methods implicitly require that objects and arrays be **non-null**.
- We extend that prohibition to the reps of abstract data types. By default, the rep invariant implicitly includes **`x != null`** for every reference **`x`** in the rep that has object type (including references inside arrays or lists).
- Although you don't need to state it in a rep invariant comment, you still must implement the **`x != null`** check, and make sure that your **`checkRep()`** correctly fails when **`x`** is null.



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**.

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, $AF(\text{numerator}, \text{denominator}) = \text{numerator}/\text{denominator}$.
- Another way of thinking about it is that the 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.

Why is beneficent mutation required?

- This kind of implementer freedom often permits performance improvements like:
 - Caching
 - Data structure rebalancing
 - Lazy computation
 - Lazy cleanup



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.
- Another piece of documentation is a **rep exposure safety argument** . 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.

Documenting AF and RI: 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, underscore
s)
    //   text.length <= 140
    // Abstraction Function:
    //   represents 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 client
s.

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

Documenting AF and RI: 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 { ... }
    ...
}
```

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.
- **Translating this in terms of the types of ADT operations, this means:**
 - Creators and producers must establish the invariant for new object instances;
 - Mutators and observers must preserve the invariant.

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.
- So the full rule for proving invariants is: **Structural induction** --- 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.



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.

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



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.



The end

March 20, 2018