A data abstraction is defined by a specification

A collection of procedural abstractions
– Not a collection of procedures

Together, these procedural abstractions provide some set of values
All the ways of directly using that set of values
– Creating
– Manipulating
– Observing

• Creators and producers: make new values
• Mutators: change the value (but don’t affect ==)
• Observers: allow one to distinguish different values

Connecting implementations to specs

Representation Invariant: maps Object → boolean
– Indicates if an instance is well-formed
– Defines the set of valid concrete values
– Only values in the valid set make sense as implementations of an abstract value
– For implementors/debuggers/maintainers of the abstraction: no object should ever violate the rep invariant
  • Such an object has no useful meaning

Abstraction Function: maps Object → abstract value
– What the data structure means as an abstract value
– How the data structure is to be interpreted
– Only defined on objects meeting the rep invariant
– For implementors/debuggers/maintainers of the abstraction:
  Each procedure should meet its spec (abstract values) by “doing the right thing” with the concrete representation

Example: CharSet Abstraction

// Overview: A CharSet is a finite mutable set of Characters
// @effects: creates a fresh, empty CharSet
public CharSet() {...}
// @modifies: this
// @effects: this_post = this_pre + {c}
public void insert(Character c) {...}
// @modifies: this
// @effects: this_post = this_pre - {c}
public void delete(Character c) {...}
// @return: (c ∈ this)
public boolean member(Character c) {...}
// @return: cardinality of this
public int size() {...}
An implementation: Is it right?

```java
class CharSet {
  private List<Character> elts = new ArrayList<Character>();
  public void insert(Character c) {
    elts.add(c);
  }
  public void delete(Character c) {
    elts.remove(c);
  }
  public boolean member(Character c) {
    return elts.contains(c);
  }
  public int size() {
    return elts.size();
  }
}
```

CharSet s = new CharSet();
Character a = new Character('a');
s.insert(a);
s.insert(a);
s.delete(a);
if (s.member(a)) {
  System.out.print("wrong");
} else {
  System.out.print("right");
}

Where is the error?

- Answer this and you know what to fix
- Perhaps `delete` is wrong
  - Should remove all occurrences?
- Perhaps `insert` is wrong
  - Should not insert a character that is already there?
- How can we know?
  - The representation invariant tells us
  - If it's "our code", this is how we document our choice for "the right answer"

Where Is the Error?

- Answer this and you know what to fix

The representation invariant

- Defines data structure well-formedness
- Must hold before and after every CharSet operation
- Operations (methods) may depend on it
- Write it like this:

  ```java
class CharSet {
  // Rep invariant:
  //   elts has no nulls and no duplicates
  private List<Character> elts = ...
  ...
  }
```

Or, more formally (if you prefer):

- ∀ indices i of `elts`
- `elts.elementAt(i)` ≠ null
- ∀ indices i, j of `elts`
- i ≠ j ⇒ ¬ `elts.elementAt(i).equals(elts.elementAt(j))`

Now we can locate the error

```java
// Rep invariant:
//   elts has no nulls and no duplicates

public void insert(Character c) {
  elts.add(c);
}

public void delete(Character c) {
  elts.remove(c);
}
```

Another example

```java
class Account {
  private int balance;
  // history of all transactions
  private List<Transaction> transactions;
  ...
}
```

Real-world constraints:
- Balance ≥ 0
- Balance = Σ transactions.get(i).amount

Implementation-related constraints:
- Transactions ≠ null
- No nulls in transactions

Checking rep invariants

Should code check that the rep invariant holds?

- Yes, if it’s inexpensive [depends on the invariant]
- Yes, for debugging [even when it’s expensive]
- Often hard to justify turning the checking off
- Some private methods need not check (Why?)

A great debugging technique:

- Design your code to catch bugs by implementing and using rep-invariant checking
Checking the rep invariant

Rule of thumb: check on entry and on exit (why?)

```
public void delete(Character c) {
    checkRep();
    elts.remove(c);
    // Is this guaranteed to get called?
    // (could guarantee it with a finally block)
    checkRep();
}
```

/** Verify that elts contains no duplicates. */
private void checkRep() {
    for (int i = 0; i < elts.size(); i++) {
        assert elts.indexOf(elts.elementAt(i)) == i;
    }
}

Listing the elements of a CharSet

Consider adding the following method to CharSet

```
// returns: a List containing the members of this
public List<Character> getElts();
```

Consider this implementation:

```
// Rep invariant: elts has no nulls and no dups
public List<Character> getElts() { return elts; }
```

Does the implementation of getElts preserve the rep invariant?
Kind of, sort of, not really....

Avoiding representation exposure

The first step for getting help is to recognize you have a problem 😊

- *Understand* what representation exposure is
- *Design* ADT implementations to make sure it doesn’t happen
- Treat rep exposure as a bug: fix your bugs
- *Test* for it with adversarial clients:
  - Pass values to methods and then mutate them
  - Mutate values returned from methods

Practice defensive programming

- Assume that you will make mistakes
- Write and incorporate code designed to catch them
  - On entry:
    - Check rep invariant
    - Check preconditions
  - On exit:
    - Check rep invariant
    - Check postconditions
- Checking the rep invariant helps you *discover* errors
- Reasoning about the rep invariant helps you *avoid* errors

Representation exposure

Consider this client code (outside the CharSet implementation):

```
CharSet s = new CharSet();
Character a = new Character('a');
s.insert(a);
s.getElts().add(a);
s.delete(a);
if (s.member(a)) …
```

- Representation exposure is external access to the rep
- Representation exposure is almost always *evil*
  - *A big deal. a common bug. you now have a name for it!*
- If you do it, document why and how
  - And feel guilty about it!

private is not enough

- Making fields *private* does *not* suffice to prevent rep exposure
  - See our example
  - Issue is aliasing of mutable data inside and outside the abstraction

- So *private* is a hint to you: no aliases outside abstraction to references to mutable data reachable from *private* fields
- Two general ways to avoid representation exposure…
Avoiding rep exposure (way #1)

- One way to avoid rep exposure is to make copies of all data that cross the abstraction barrier
  - Copy in [parameters that become part of the implementation]
  - Copy out [results that are part of the implementation]

- Examples of copying (assume Point is a mutable ADT):
  ```java
class Line {
  private Point s, e;
  public Line(Point s, Point e) {
    this.s = new Point(s.x, s.y);
    this.e = new Point(e.x, e.y);
  }
  public Point getStart() {
    return new Point(this.s.x, this.s.y);
  }
  ...
}
```

Need deep copying

- “Shallow” copying is not enough
  - Prevent any aliasing to mutable data inside/outside abstraction

- What’s the bug (assuming Point is a mutable ADT)?
  ```java
class PointSet {
  private List<Point> points = ...
  public List<Point> getElts() {
    return new ArrayList<Point>(points);
  }
}
```

- Not in example: Also need deep copying on “copy in”

Avoiding rep exposure (way #2)

- One way to avoid rep exposure is to exploit the immutability of (other) ADTs the implementation uses
  - Aliasing is no problem if nobody can change data
    - Have to mutate the rep to break the rep invariant

- Examples (assuming Point is an immutable ADT):
  ```java
class Line {
  private Point s, e;
  public Line(Point s, Point e) {
    this.s = s;
    this.e = e;
  }
  public Point getStart() {
    return this.s;
  }
  ...
}
```

Why [not] immutability?

- Several advantages of immutability
  - Aliasing does not matter
  - No need to make copies with identical contents
  - Rep invariants cannot be broken
  - Cf. CSE341 for more!

- Does require different designs (e.g., if Point immutable)
  ```java
  void raiseLine(double deltaY) {
    this.s = new Point(s.x, s.y+deltaY);
    this.e = new Point(e.x, e.y+deltaY);
  }
  ...
}
```

- Immutable classes in Java libraries include String, Character, Integer, ...

Deepness, redux

- An immutable ADT must be immutable “all the way down”
  - No references reachable to data that may be mutated

- So combining our two ways to avoid rep exposure:
  - Must copy-in, copy-out “all the way down” to immutable parts

Back to getElts

Recall our initial rep-exposure example:
```java
class CharSet {
  // Rep invariant: elts has no nulls and no dups
  private List<Character> elts = ...;

  // returns: elts currently in the set
  public List<Character> getElts() {
    return new ArrayList<Character>(elts); //copy out!
  }
  ...
}
```
An alternative

```java
// returns: elts currently in the set
public List<Character> getElts() { // version 1
    return new ArrayList<Character>(elts);//copy out!
}

public List<Character> getElts() { // version 2
    return Collections.unmodifiableList<Character>(elts);
}
```

From the JavaDoc for `Collections.unmodifiableList`:

Returns an unmodifiable view of the specified list. This method allows modules to provide users with "read-only" access to internal lists. Query operations on the returned list "read through" to the specified list, and attempts to modify the returned list… result in an `UnsupportedOperationException`.

The good news

```java
public List<Character> getElts() { // version 2
    return Collections.unmodifiableList<Character>(elts);
}
```

- Clients cannot modify (mutate) the rep
- So they cannot break the rep invariant
- (For long lists,) more efficient than copy out
- Uses standard libraries

The bad news

```java
public List<Character> getElts() { // version 1
    return new ArrayList<Character>(elts);//copy out!
}

public List<Character> getElts() { // version 2
    return Collections.unmodifiableList<Character>(elts);
}
```

The two implementations do not do the same thing!
- Both avoid allowing clients to break the rep invariant
- Both return a list containing the elements

But consider:
```
xs = s.getElts();
s.insert('a');
xs.contains('a');
```

Version 2 is observing an exposed rep, leading to different behavior

Different specifications

Ambiguity of "returns a list containing the current set elements"

- "returns a fresh mutable list containing the elements in the set at the time of the call"
- versus
  - "returns read-only access to a list that the ADT continues to update to hold the current elements in the set"

A third spec weaker than both [but less simple and useful!]

- "returns a list containing the current set elements. Behavior is unspecified (!) if client attempts to mutate the list or to access the list after the set's elements are changed"

Also note: Version 2's spec also makes changing the rep later harder
- Only "simple" to implement with rep as a `List`