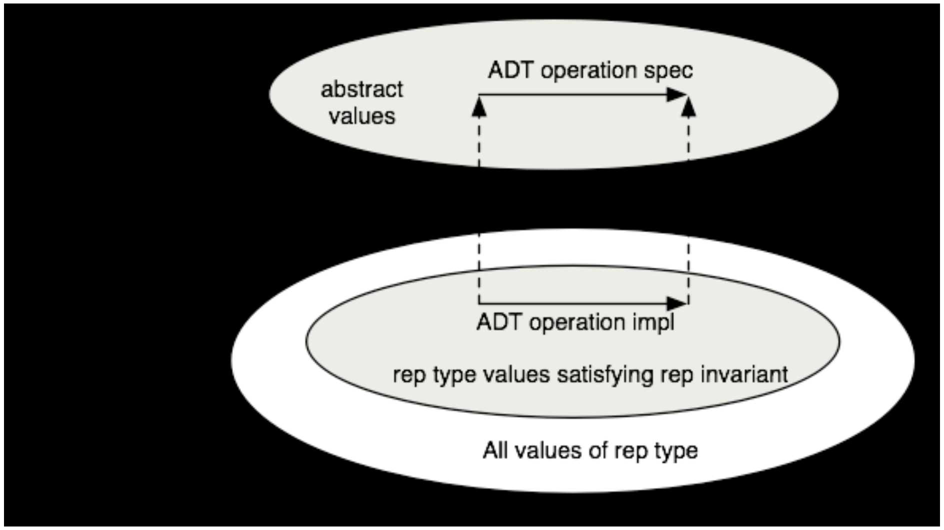
Representation Invariants and Abstraction Functions



Designing Data Structures

- From domain concept
  - E.g., the math concept of a polynomial, an integer set, the concept of a library item, etc.
- through ADT
  - Describes domain concept in terms specification fields and abstract operations
- to implementation
  - Implements ADT with representation fields and concrete operations

Specifying an ADT

immutable

class TypeName

1. overview
2. specification fields
3. creators
4. observers
5. producers
6. ~~mutators~~

mutable

class TypeName

1. overview
2. specification fields
3. creators
4. observers
5. producers (rare)
6. mutators

Example: Python Data Types

immutable

Tuple (1, “cat”)

1. overview

Creator

2. specification fields
3. Creators x = (1, "cat")
4. observers x[1], len(x)
5. Producers z = (2, "dog"); w = x + z
6. ~~mutators~~

Producer

mutable

List [1,2,3,4, 5]

1. overview
2. specification fields
3. creators y = [1,2,3, 4, 5]
4. observers y[1:3]
5. producers t = y + [7,8]
6. mutators y.append(6)

IntSet - One Possible Implementation

```
class IntSet {
   // Rep Invariant: data contains no duplicates and no nulls
   private List<Integer> data = new ArrayList<Integer>();
   public void add(Integer x) {
         if(! contains(x))
                  data.add(x);
   }
   public void remove(Integer x) {
       data.remove(x);
   }
   public boolean contains(Integer x) {
       return data.contains(x);
   }
   public int size() { return data.size(); };
   public List<Integer> getElements() { return data;}
}
```

Representation Exposure

- Client can get control over rep and break the rep invariant! Consider

```
IntSet s = new IntSet();
s.add(27);
List<Integer> li = s.getElements();
li.remove(0); // alters , remove element 0
```

- Representation exposure is unintentional external access to the underlying representation of an object.
  - Allows access without going through object's public methods
  - Representation exposure can cause problems.
- If you must allow representation exposure to a mutable object, document why and how and *feel bad about it*

Representation Exposure

- Make a copy on the way out:

```
public List<Integer> getElements() {
  return new ArrayList<Integer>(data);
}
```

- Mutating a copy does not affect **`IntSet`**'s rep

```
IntSet s = new IntSet();
s.add(1);
List<Integer> li = s.getElements();
li.remove(1); //mutates new copy, not s's rep
```

Integer is immutable. Does this make a difference?

Representation Exposure

- Make a copy on the way in too:

```
public IntSet(ArrayList<Integer> elts) {
  data = new ArrayList<Integer>(elts);
  …
}
```

- Why?

Representation Exposure

- What if we made a StringSet like IntSet, do we have to worry about rep exposure?

- Returning a primitive like an int, float, etc. does not cause a dangerous rep exposure.
  - Primitives are copied when used with a return statement.

Representation Exposure

- How about this:

```
class Movie {
  private String title;
  …
  public String getTitle() {
    return title;
  }
}
```

- Technically, there is representation exposure
- Representation exposure is dangerous when the rep is mutable
  - If the rep is immutable, it's OK

Immutability, again

- Suppose we add an iterator

// returns: an Iterator over the IntSet

```
public Iterator<Integer> iterator();
```

- Suppose the following implementation:

```
public Iterator<Integer> iterator() {
   return data.iterator();
}
```

Is there a possible problem?

An iterator to a mutable object like an ArrayList can allow client to modify object.

Immutability, again

```
class IntIterator {
  private List<Integer> theData;
  private int next;

  public IntIterator(List<Integer> data) {
    // make a copy of the data
    theData = new ArrayList<Integer>(data);
    next = 0;
  }
  public boolean hasNext() {
    return (next < theData.size());
  }
  public int next() {
    return theData.get(next++);
  }
}
```

Rep Exposure Can Be Subtle

```
public class PointSet {
        // Rep Invariant: data contains no duplicates and no nulls
        // Point is mutable
        private List<Point> data = new ArrayList<Point>();

        public void add(Point p) {
                if(! contains(p))
                        data.add(p);
        }
        public void remove(Point p) {
                data.remove(p);
        }
        public boolean contains(Point p) {
                return data.contains(p);
        }
        public int size() { return data.size(); };

        public List<Point> getElements() { return new ArrayList(data);}

}
```

What Happens Here?

```
Point p1 = new Point(1,2);
Point p2 = new Point(3,3);

PointSet ps = new PointSet();
ps.add(p1);
ps.add(p2);
ps.print();

List<Point> lp = ps.getElements();
lp.remove(p1);
ps.print();  // safe

p1.setY(57);
ps.print();  // Huh?!
```

Rep Exposure of Mutable Elements is a Problem

- lp.remove(p1) is safe
  - Removes an element from lp list
- ArrayList.copy() makes a copy of each element
  - Copies references, doesn't make a copy of the data
  - remove() finds item to remove by reference equality
    - .equals has not been overriden

- p1.setY(57); ps.print(); // Huh?!
- p1 is a reference to a mutable point
- By changing p1, we change object on heap
  - data ArrayList contains a reference to that object
  - ArrayList contains a reference to mutated object
- We can change the representation

- Moral: be careful with mutable objects
  - Prefer immutable objects whenever possible

Checking Rep Invariant

- **`checkRep()`** or **`repOK()`**
- Always check if rep invariant holds when debugging
- Leave checks in production code, if they are inexpensive
- Checking rep invariant of **`IntSet`**

```
// throws NullPointerException if data contains a null
private void checkRep() {
  for(d : data)
     if(d == null)
         throw new NullPointerException("null data");

  for (int i=0; i<data.size; i++) {
    if (data.indexOf(data.elementAt(i)) != i)
      throw new RuntimeException("duplicates!");
  }
}
```

Checking Rep Invariant – different way

```
private void checkRep() {
      for(d : data)
         if(d == null)
            throw new NullPointerException("null data");

      Set<Integer> set = new HashSet<Integer>(data);
      if(set.size() != data.size())
           throw new RuntimeException("duplicates!");
}
```

Practice Defensive Programming

- Assume that you will make mistakes
- Write code to catch them
  - On method entry
    - Check rep invariant (i.e., call checkRep())
      - Can help find rep exposure
    - Check preconditions (requires clause)
  - On method exit
    - Check rep invariant (call checkRep())
    - Check postconditions
- Checking rep invariant helps find bugs
- Reasoning about rep invariant helps avoid bugs

Aside: Practice Defensive Programming

- https://www.youtube.com/watch?v=C_r5UJrxcck

Aside: Invariants

- Why focus so much on invariants?
  - Loop invariants, rep invariants, immutability,
    - Immutability is a kind of invariant
    - immutable ADTs;
    - modifies and effects clauses in the specification are empty

- Software is complex
  - Lots of interactions between different “modules”
  - Interactions make reasoning difficult
  - Lots of “moving parts” (i.e., lots of changes)

Invariants

- Invariants are properties that stay unchanged
  - Reduces intellectual complexity
  - Reduces cognitive burden

- If we know that some property stays unchanged, we can focus on other properties

- Reducing the number of things we need to think about can be of great benefit

Connecting Implementation to Specification

- Representation invariant: Object → boolean
  - Indicates whether data representation is well-formed. Only well-formed representations are meaningful
  - Defines the set of valid values
- Abstraction function: Object → abstract value
  - What the data representation really means
    - E.g., array [2, 3, -1] represents $-x^2 + 3x + 2$
  - How the data structure is to be interpreted

Abstraction Function: rep → abstract value

- The abstraction function maps valid concrete data representation to the abstract value it represents.
  - I.e., domain is all reps that satisfy the rep invariant
  - Range is the abstract value represented
- AF: Object → abstract value

- The abstraction function lets us reason about behavior from the client perspective

Abstraction Function Example

```
class Poly {
   // Rep invariant: degree = coeffs.length-1
   //                coeffs[degree] != 0
   private int[] coeffs;
   private int degree;
   // Abstraction function: coeffs [a0,a1,…adegree]
   // represents polynomial
   // adegree x^degree + … + a1x + a0
   // E.g., array [-2,1,3] → 3x^2 + x - 2

   // Empty array represents the 0 polynomial

   …
```

Another Abstraction Function Example

```
class IntSet {
   // Rep invariant:
   // data contains no nulls and no duplicates
   private List<Integer> data;
  // Abstraction function: data [a1,a2,…an]
   // represents the set { a1, a2, … an }.
   // Empty list represents {}.

   …
   public IntSet() …
```

Abstraction Function: mapping rep to abstract value

- Abstraction function: Object → abstract value
  - Maps the concrete representation to the abstract representation
  - I.e., the object's rep maps to abstract value
    - IntSet e.g.: list [2, 3, 1] → { 1, 2, 3 }
  - Many objects map to the same abstract value
    - IntSet e.g.: [2, 3, 1] → { 1, 2, 3 } and [3, 1, 2] → { 1, 2, 3 } and [1, 2, 3] → { 1, 2, 3 }
- Not a function in the opposite direction
  - Different representation values can map to the same abstract value
  - abstract value -> object is a class.

Another (Implementation of) IntSet

- What if we dropped the “no duplicates” constraint from the rep invariant

```
class IntSet {
 // Rep invariant: data contains no nulls
     private List<Integer> data;
   …
```

- Can we still represent the concept of the IntSet? (Remember, an IntSet is a mutable set of integers with no duplicates.)

Yes. First, we have to change the abstraction function

```
class IntSet {
   // Rep invariant: data contains no nulls
   private List<Integer> data;
  // Abstraction function: List data
   // represents the smallest set
   // { a1, a2, … an } such that each ai is
   // in data. Empty list represents {}.

   …
   public IntSet() …
```

Another IntSet

```
class IntSet {
      // Rep invariant: data contains no nulls
      private List<Integer> data;
   …
```

- [1,1,2,3] → { 1, 2, 3 }
- [1,2,3,1] → { 1, 2, 3 }

etc.

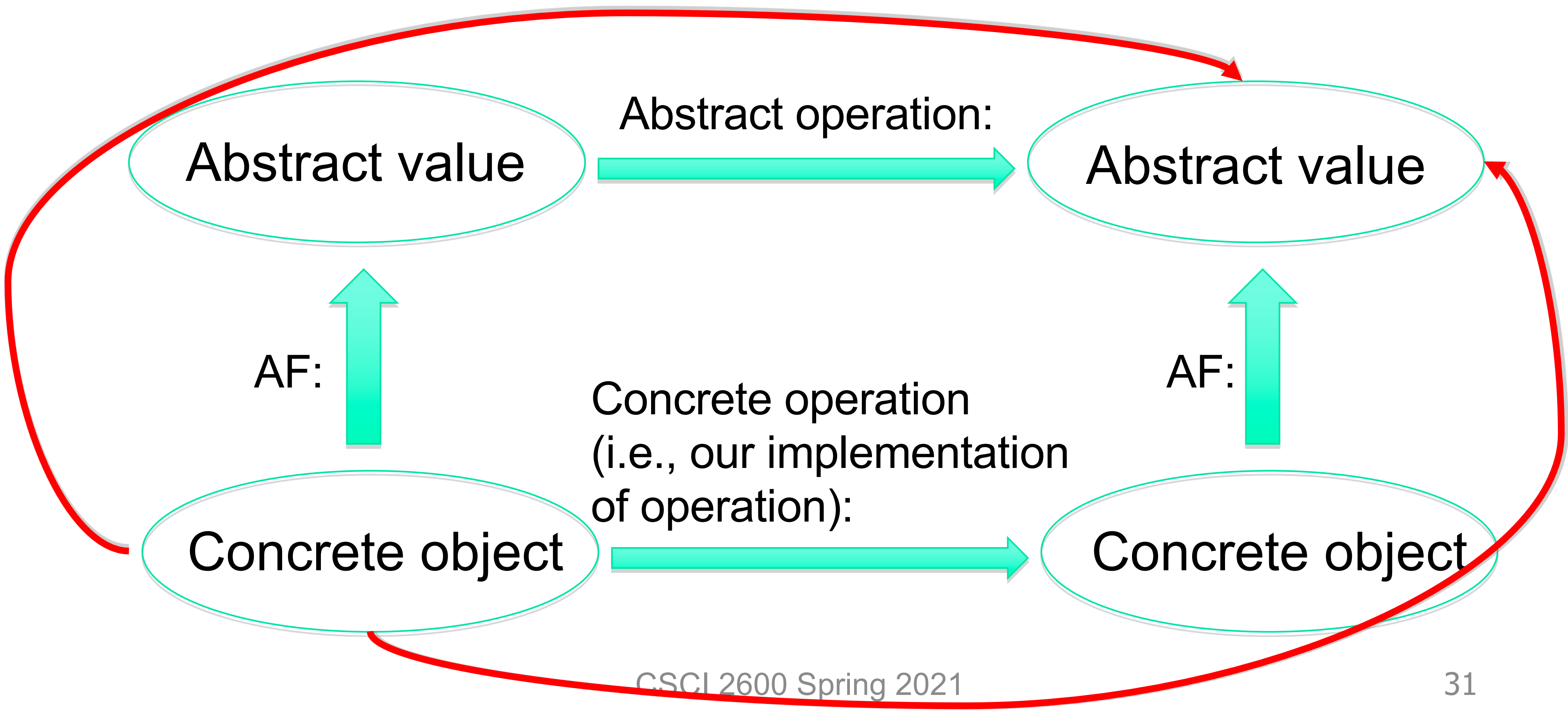
There are many objects that correspond to the same abstract value

Another IntSet

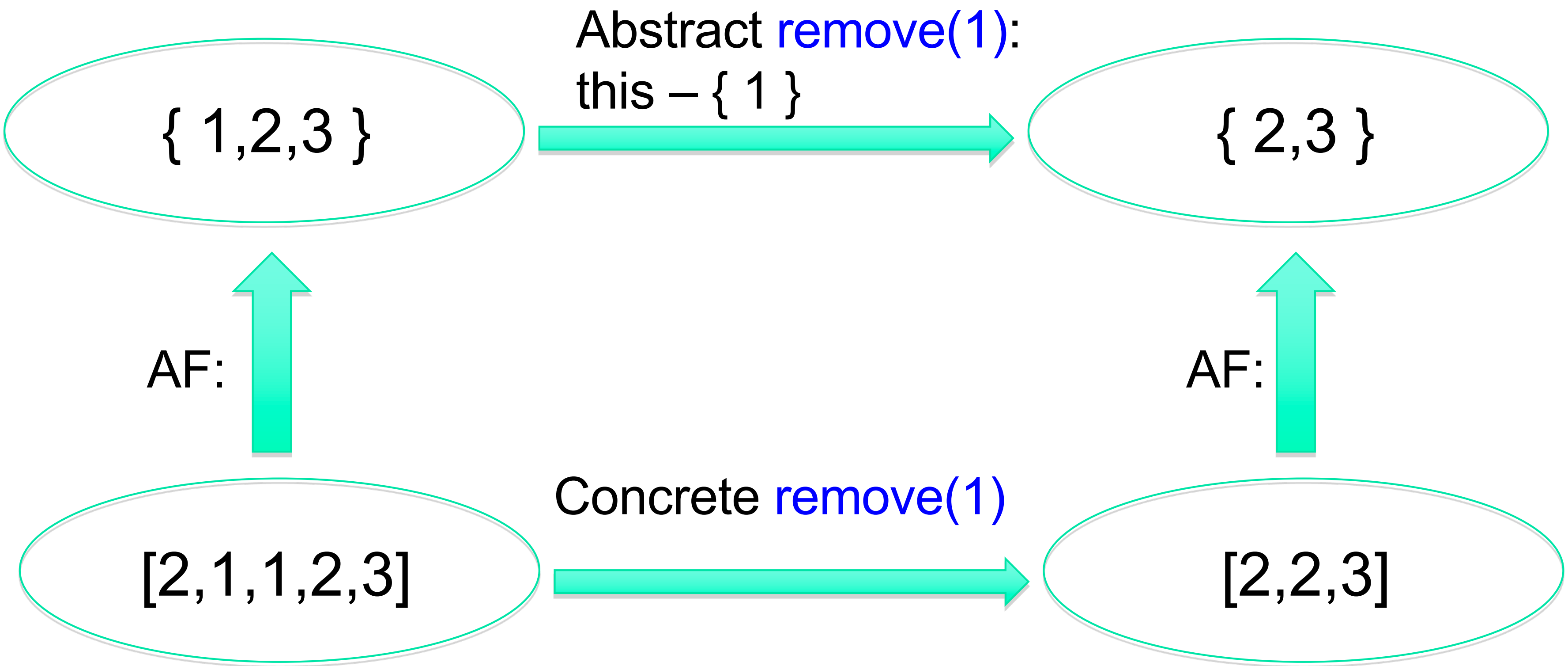
- We have to change the implementation of operations as well
- What is the implication for **`add(int x)`** and **`remove(int x)`**? For **`size()`**?
- **`add(int x)`** no longer needs check **`contains(x)`**. Why?
- **`remove(int x)`** must remove all occurrences of **`x`** in **`data`**. Why?
- What about print**`()`**? What else?

Correctness

- Abstraction function allows us to reason about the implementation



IntSet Example



Creating concrete object:
- Establish rep invariant
- Establish abstraction function

After every operations:
- Maintains rep invariant
- Maintains abstraction function

Aside: the Rep Invariant

- Which implementation is better

```
class IntSet {
 // Rep invariant:
 // data has no nulls and no duplicates
     …
 // methods establish & maintain invariant
 // and original abstraction function
```

or

```
class IntSet {
 // Rep invariant: data has no nulls
     …
 // methods maintain this weaker invariant
 // and new abstraction function
```

Aside: the Rep Invariant

- Often one role of the rep invariant is to <u>simplify</u> the abstraction function (by limiting valid concrete values which limits the domain of the abstraction function)

- Consequently, rep invariant simplifies implementation and reasoning!

Aside: Benevolent Side Effects

- Another implementation of IntSet.contains:

```
boolean contains(int x) {
   int i = data.indexOf(x);
   if (i == -1)
      return false;
   // move-to front optimization
   // speeds up repeated membership tests
   int y = data.elementAt(0);
   data.set(0,x);
   data.set(i,y);
}
```

- <u>Mutates rep, but does not change abstract value</u>!

Writing an Abstraction Function

- The domain is all representations that satisfy the rep invariant
  - Rep invariant simplifies the AF by restricting its domain
- The range (set of abstract values) can be tricky to denote
  - Relatively easy for mathematical concepts like sets
  - Trickier for “real-world” ADTs
    - Use specification fields and derived specification fields to describe abstract values

Specification Fields

- Describe abstract values. Use in overview of ADT. Think of the abstract value as if it were an object with fields
- Polynomial abstraction:
  - Univariate polynomial $a_n x^n + \ldots + a_1 x + a_0$
  - $a_n, \ldots\ a_1,\ a_0$ are specification fields
  - **`degree`** is a derived specification field
- Define AF and specs of abstract operations in terms of specification fields

Specification Fields

- Often abstract values aren't clean mathematical objects
  - E.g., concept of **`Customer`**, **`Meeting`**, **`Item`**
  - Define those in terms of specification fields: e.g., a **`Meeting`** can be specified with specification fields **`date, location, attendees`**
- In general, the specification fields (the specification) are different from the representation fields (instance fields in the implementation)

ADTs and Java Language Features

- Java classes
  - Make operations of the ADT public methods
  - Make other operations private
  - Clients can only access the ADT operations
- Java interfaces
  - Clients only see the ADT operations, nothing else
  - Multiple implementations, no code in common
  - Cannot include creators (constructors) or fields

ADTs and Java Language Features

- Both classes and interfaces rely upon careful specifications
- Prefer interface types instead of specific classes
  - e.g., we used **List<Integer>** as the type of the **data** field, not **ArrayList<Integer>**
  - Why?
    - This is preferred because you decouple your code from the implementation of the list
    - https://www.javaworld.com/article/2073649/core-java/why-extends-is-evil.html

Exercise

- Write the abstraction function for the mathematical concept of the **`Line`**
  - Choose specification fields (abstraction)
  - Choose representation fields
  - Write rep invariant
  - Write abstraction function

Spec fields: point1(x,y), point2(x,y)
Rep fields: x1,y1,x2,y2
Rep invariant: x1 != x2 || y1 != y2
AF: A line segment is a pair of 2D points x, y s.t. point1 != point2

Exercise

- Suppose we decided to represent our polynomial with a list of terms:
- `private List<Terms> terms;`
- Write the abstraction function
  - `terms.getExp(i)` refers to the exponent of i^{th} term
  - `terms.getCoef(i)` refers to the coefficient of i^{th} term

AF(r) = a polynomial s.t. the collection

```
<terms.getCoef(0), terms.getExp(0)> …
   <terms.getCoef(degree), terms.getExp(degree)>
```

$\rightarrow a_{degree}x^{degree} + \ldots + a_1 x + a_0$

```
E.g., array [Term(-2,0), Term(1,1), Term(3,2)]
```

$\rightarrow 3x^2 + x - 2$

Implementing an ADT: Summary

- Rep invariant
  - Defines the set of valid objects (concrete values)
- Abstraction function
  - Defines, for each valid object, which abstract value it represents
- Together they modularize the implementation
  - Can reason about operations in isolation
  - Neither is part of the ADT abstraction!!!

Implementing an ADT: Summary

- In practice
  - Always write a rep invariant!
  - Write an abstraction function when you need it
      - A description is important
    - Write a precise and concise, but relatively informal abstraction function
    - A formal one is hard to write, and often not that useful
    - As always with specs: we look for the balance between what is “formal enough to do reasoning” and what is “humanly readable and useful”