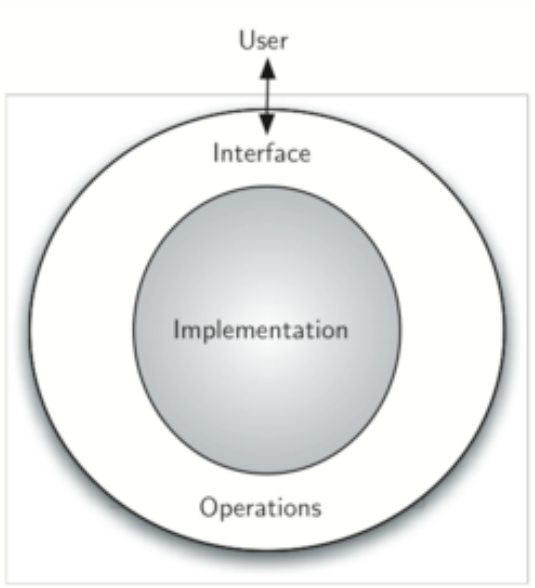
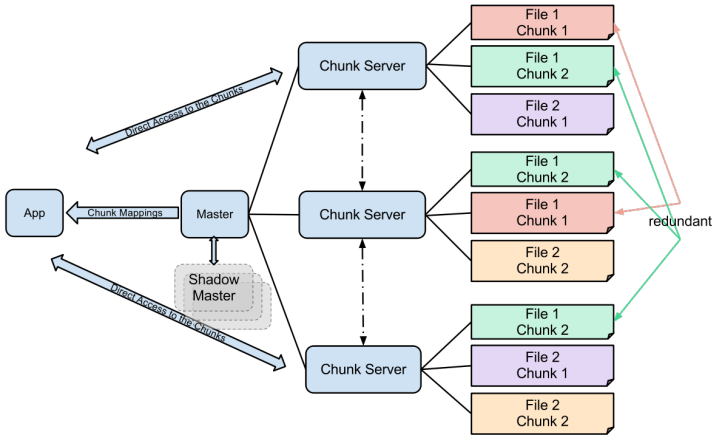
Abstract Data Types (ADTs)



Abstraction

- Abstraction: hiding unnecessary low-level details
  - An abstraction is a simplified view of an entity, which omits *unimportant* details
- An abstraction can go wrong if
  - It includes details that are not important
    - Increases the cognitive load of the programmer
  - It omits important details
    - Programmer doesn't have necessary information to use the module, class, function, etc.

Abstraction



- A file system is an abstraction
  - Omits many details, such as the mechanism for choosing which blocks on a storage device to use for the data in a given file
  - Most users don't care about these details
  - Some of the details of a file system's implementation are important to some users
    - Most file systems cache data in main memory, and they may delay writing new data to the storage device in order to improve performance.
    - Some applications, such as databases, need to know exactly when data is written through to storage, so they can ensure that data will be preserved after system crashes.
    - Thus, the rules for flushing data to secondary storage must be visible in the file system's interface.
  - Abstraction should provide appropriate detail to appropriate users

Control Abstraction

- Control abstraction (procedural abstraction)
  - A procedure (method) implements the details of an algorithm
  - One part of abstraction: signature, provides name, parameter types, return type.
    - E.g., **int binarySearch(int[] a, int key)**
  - Another part: specification, provides details about behavior and effects
  - Reasoning about code connects implementation to specification

Data Abstraction

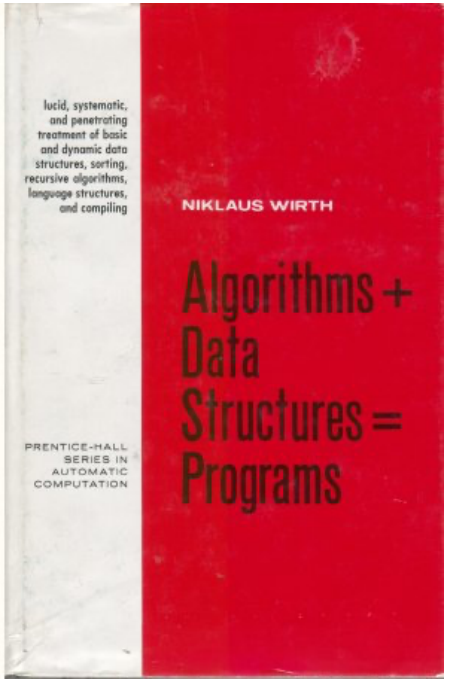
- Data abstraction
  - Types: abstract away from the details of data representation
  - E.g., type String is an abstraction
  - E.g., C type struct Person is an abstraction
- Abstract Data Type (ADT): high-level data abstraction
  - The ADT is operations + state
  - A specification mechanism
  - A way of thinking about programs and design

Abstract Data Types are Important

- ADTs are about organizing and manipulating data
- Organizing and manipulating data is pervasive.
    - Inventing and describing algorithms is not
- Start your design by thinking about the data model.
    - What is the data?
    - Consider what operations are needed.
    - Implementation
        - Chose data structures carefully.
            - Will they allow necessary operations to be efficient?
        - Write code to access and manipulate data.

Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.

~ Rob Pike



ADT is a way of thinking about programs and design

- From domain concept
  - E.g., the math concept of a polynomial, a set of integers, the concept of a library item, etc.
    - abstract
- through ADT
  - Describes domain concept in terms of specification fields and abstract operations
    - Abstract
    - Specification fields – what are the important pieces of information?
    - Operations – what manipulations will be needed
- to implementation
  - Implements ADT with representation fields and concrete operations
    - Concrete
- Abstraction is how programmers deal with complexity

Example: Polynomial with Integer Coefficients - ADT

ADT:

Overview description:

A Poly is an immutable polynomial with integer coefficients. A Poly is:

$c_0 + c_1x + c_2x^2 +$ …

- **Specification fields c_0, c_1 etc.**

Set of abstract operations:

add, mul, eval, etc. with PSoft style specs referencing abstract specification fields

Example: Polynomial with Integer Coefficients - Implementation

```
class Poly {
    // rep. invariant: d = coeffs.length-1
    private int d; // degree of the polynomial
    private int[] coeffs; // coefficients
    // operations add, sub, mul,
    // eval, in terms of rep. fields coeffs, d.
}
```

```
class Poly {
    // rep. invariant: …
    private List<Term> terms; // terms of poly
    // operations add, sub, mul, eval, etc. in terms of terms
    // term – object specifying coefficient and exponent
}
```

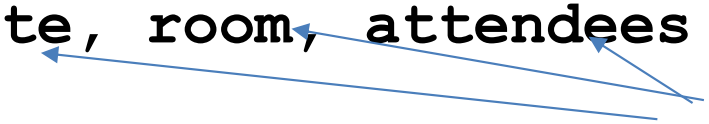
Another Example: A Meeting: Domain Concept & ADT

ADT:

Overview description:

`An appointment for a meeting.`

`With date, room, attendees`

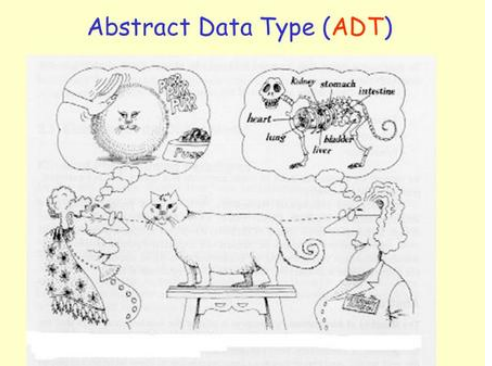


Specification fields.

Set of abstract operations:

`e.g., addAttendee, scheduleRoom, etc.` with PSoft style specs referencing abstract specification fields

Why ADTs?



- Bridges gap between domain concept and implementation
- Formalizes domain concept, provides basis for reasoning about correctness of the implementation
- **Shields client from implementation. Implementation can vary without affecting client!**
  - Information hiding
  - Client code interacts with object through a set of abstract operations

An ADT has a Set of Operations

- Operations act on data
- ADT is about the meaning of data
- ADT is about the use of data
- Data representation (implementation) may change without causing problems for the client.

```
class RightTriangle {
 float base, altitude;
}
```

```
class RightTriangle {
 float base, angle;
}
```

- Instead, think of a type as a set of operations: create, getBase, getAltitude, getBottomAngle, etc.
- Force clients to call operations to access data
  - Client doesn't need to know the details of representation

Are These Types Same or Different?

```
class Point {
 float x;
 float y;
}
```

```
class Point {
 float r;
 float theta;
}
```

- They are different! Different implementations
- They are the same! Both implement the concept of a 2D point.
- Goal of ADT methodology is to express sameness
  - Clients depend only on the set of operations: x(), y(), r(), theta(), etc.
  - Data representation can be changed: to change algorithms, to fix bugs, etc.

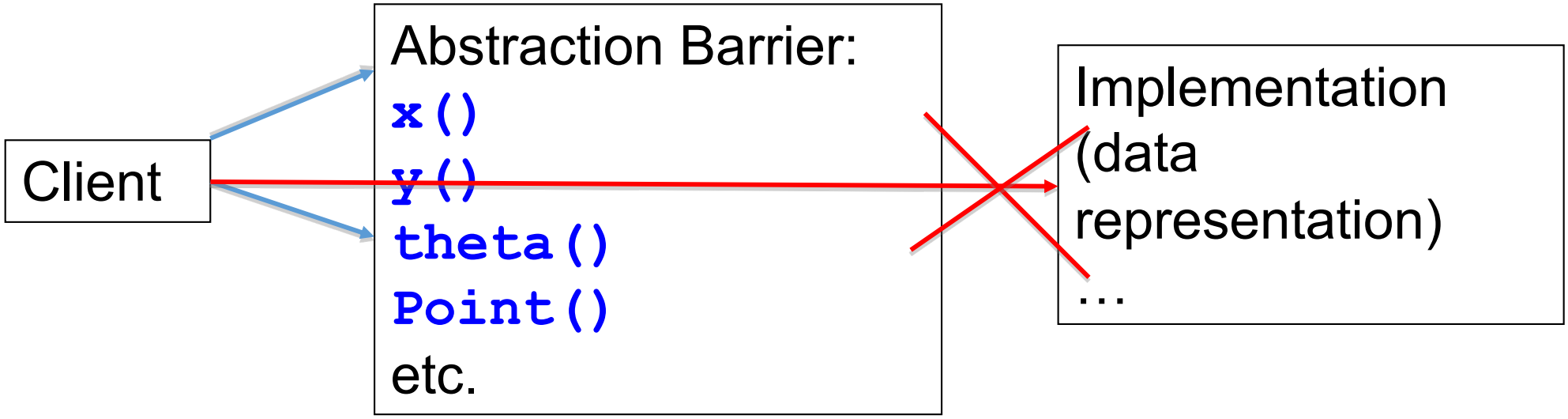
Are These Types Same or Different?

```
class Poly {
   private int d;
   private int[] coeffs;
}
```

```
class Poly {
   private List<Term> terms;
}
```

- Same ADT
- Clients depend only on the set of operations: **`add(Poly)`**, **`mul(Poly)`**, etc.

Abstraction Barrier



ADT provides an abstraction barrier
Clients access the ADT through its operations.
They never access the data representation.

ADT Methods

- We group the access methods of an ADT into
  - Creators
    - Create a brand new object
    - Constructors are creators
  - Observers
    - Return information about *this* object
  - Producers
    - Return a new object of *this* type by performing operations on the current object
    - Poly add(Poly p) – return a new Poly, this + p
  - Mutators
    - Change the current object

2D Point abstract methods

```
class Point {
   // A 2D point in the plane
   public float x();
   public float y();
   public float r();
   public float theta();

   // … can be created
   public Point(); // (0,0)
   public Point(float x, float y);

   public Point swapXY();
```

Observers

Creators

Producer

2D Point as an ADT

```
// class Point continued
   …

   // … can be moved
   public void translate(float delta_x,
                         float delta_y);
   public void scaleAndRotate(float delta_r,
                              float delta_theta);

}
```

Mutators

Specifying an ADT

immutable

class TypeName

1. overview
2. specification fields
3. creators
4. observers
5. producers
6. ~~no mutators~~

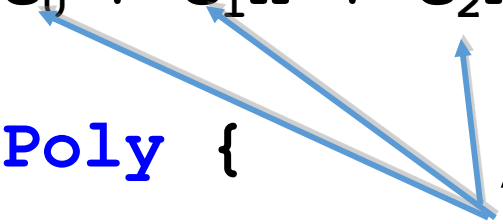
mutable

class TypeName

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

Poly, an immutable datatype: overview

```
/**
* A Poly is an immutable polynomial with
* integer coefficients. A Poly is:
*      c0 + c1x + c2x^2 + ...
*/
class Poly {
```



Abstract state (specification fields).
More on this later.

Overview: Always state whether mutable or immutable

Define abstract model for use in specification of operations. In ADTs, state is abstract, not concrete

I.e., these are NOT actual, implementation fields of Poly, just what we call specification fields.

Poly, an immutable datatype: creators

```
// modifies: none
// effects: makes a new Poly = 0
public Poly()

// modifies: none
// effect: makes a new Poly = cx^n
// throws: NegExponentException if n < 0
public Poly(int c, int n)
```

Creators: This is an example of *overloading*, two **`Poly`** constructors with different signatures.

New object is part of effects, not preexisting state. Hence, modifies is none.

Poly, an immutable datatype: observers

```
// returns: degree of this polynomial
public int degree()

// returns: the coefficient of the term of
//    this polynomial, whose power is d
public int coeff(int d)
```

Observers: Used to obtain information about *this* polynomial.

Return values of other types.

Observers should **not** modify the abstract state!

this: the current Poly object. Also called the receiver

```
Poly x = new Poly(…) // creator
int c = x.coeff(3);  // observer
```

Poly, an immutable datatype: producers

```
// modifies: none
// returns: a new Poly with value this + q
public Poly add(Poly q)

// modifies: none
// returns: a new Poly with value this * q
public Poly mul(Poly q)
```

Producers: Operations on a type that create other objects of the same type. Common in immutable types.

Should have no side effects

i.e., should not change _abstract_ values of any existing object

Poly, an immutable datatype: Mutators

- Poly is an immutable ADT
- It has no mutators

IntSet, a mutable datatype: overview, creators and observers

```
/*
* Overview: An IntSet is a mutable,
* unbounded set of integers. E.g.,
*      { x1, x2 , … xn } with no duplicates
*/
class IntSet {

 // effects: makes a new empty IntSet
   public IntSet()

 // returns: true if x in this IntSet,
 //          else false
   public boolean contains(int x)
```

Creator

Observer

IntSet, a mutable datatype: mutators

```
// modifies: this
// effects: this_post = this_pre ∪ { x }
  public void add(int x)

// modifies: this
// effects: this_post = this_pre - { x }
  public void remove(int x)
```

Mutators

Mutators: operations that modify receiver, ***this***. Rarely modify anything other than ***this***.

effects: describe how ***this*** changes***.***

Often, mutators have no return value.
Although they may return true or false to indicate success

Exercise: String, an immutable datatype

- Overview?
  - an immutable sequence of characters, example “abc”
  - don’t say array or list of characters
- Creators?
  - **String(), String(char[] value), String(String original), …**
- Observers?
  - **charAt, compareTo, contains, endsWith, …**
- Producers?
  - **concat**, **format**, **substring**, …
- Mutators?
  - None!

Exercise: The Stack datatype

```
public Stack()

public boolean empty()
public E peek()
public void push(E item)
public E pop()
```

ADTs and Java Language Features

- Java classes
  - ADT operations are public
  - Other operations are private
  - Clients can only access ADT operations
- Java interfaces
  - Clients only see the ADT operations
  - Cannot see fields
  - Implementing classes must provide all operations

Reasoning About ADTs

- ADT is a specification, a set of operations
  - E.g., contains(), add(), etc., (the IntSet ADT)
  - add(Poly q), mul(Poly q), etc., (the Poly ADT)
- When specifying ADTs, there is no mention of data representation!
  - Basic information hiding principle
- When implementing the ADT, we must select a specific data representation
- Reasoning connects implementation to specification:
  - is our implementation correct?

Implementation of an ADT is Provided by a Class

- To implement the ADT
  - We must select the representation
  - Implement concrete operations in terms of that rep
  - E.g., the rep of our Poly can be
    - a) `int[] coeffs` or
    - b) `List<Term> terms`
- Choose representation such that
  - It is possible to implement all operations
  - Most frequently used operations are efficient

Connecting Implementation to Specification

- Representation invariant: Object → Boolean
  - Maps the object to a true or false value
  - Indicates whether data representation is well-formed.
    - If rep is not-well formed, operations may produce incorrect results
  - Only well-formed representations are meaningful
  - Defines the set of valid values
- Abstraction function: Object → abstract value
  - What the data structure really means
    - E.g., array [2, 3, -1] represents -x^2 + 3x + 2
  - How the data structure is to be interpreted

The Representation Invariant

- States data structure well-formedness
  - E.g., IntSet objects, whose data array contains duplicates are not valid values
- Must hold before and after every method!
  - Not necessarily in the middle of a method.
- Correctness of operation implementation (methods in the class) depends on it!

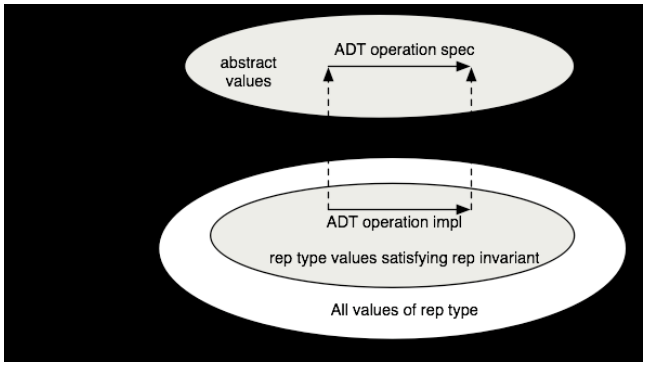
IntSet ADT

```
/** Abstraction: An IntSet is a mutable set
  * of integers with no duplicates.
  * E.g., { x1, x2 , … xn }, {}.
*/
  // rep invariant: data has no duplicates
  // effects: makes a new empty IntSet
  public IntSet();

  // modifies: this
  // effects: this_post = this_pre U { x }
  public void add(Integer x);

  // modifies: this
  // effects: this_post = this_pre - { x }
  public void remove(Integer x);

  // returns: (x in this)
  public boolean contains(Integer x)

  // returns: cardinality of this
  public int size()
```

One Possible Implementation

```
class IntSet {
    private List<Integer> data
            = new ArrayList<Integer>();
    public void add(Integer 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(); }
}
```

The Representation Invariant



- `s = new IntSet();`
  `s.add(1); s.add(1); s.remove(1);`
  `System.out.println(s.contains(1));`

- What is wrong with this code?
- Representation invariant tells us

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

The Representation Invariant

```
class IntSet {
    // Rep invariant:
    // data no duplicates
    private List<Integer>  data
            = new ArrayList<Integer>();

    public void add(Integer x) {
        data.add(x);
    }
    // Rep invariant does not hold after second
       add!
}
```

The Representation Invariant

```
class IntSet {
   // Rep invariant:
   // data has no duplicates
   private List<Integer>  data
         = new ArrayList<Integer>();
   public void add(Integer x) {
      if (!data.contains(x))
        data.add(x);
   }
   // Rep invariant now holds after add
}
```

The Representation Invariant

- Rep invariant excludes values that do not correspond to abstract values

```
class LineSegment {
  //Rep invariant: !(x1 == x2 && y1 == y2)
    private float x1, y1; // start point
    private float x2, y2; // end point
```

- Conceptually, a line segment is defined by two distinct points.
- Thus, values with the same start and end point (e.g., x1=x2=1, y1=y2=2), are meaningless.
- Rep invariant excludes them

The Representation Invariant

- Rep invariant excludes data representation values that do not correspond to abstract values

```
class RightTriangle {
    // Rep invariant: 0 < angle < 90 &
    // base > 0
float base, angle;
        // Objects that don't meet the above constraints are
        // not right triangles
```

Additionally…

- Rep invariant states constraints imposed by specific data structures and algorithms
  - E.g., Tree has no cycles, array must be sorted, at most one term per exponent in Poly (if we choose the term representation)
- Rep invariant states constraints between fields that are synchronized with each other
  - E.g., **`degree`** and **`coefficients`** fields in Poly (if we choose the array data representation)
- In general, rep invariant states correctness constraints
  - if not met, implementation may behave incorrectly!

Rep Invariant Example

```
class Account {
  // Rep invariant:
  // balance >= 0
  // balance = Σ_i transactions.get(i).amount
  // transactions != null
  // no nulls in transactions

  private int balance;
  // history of transactions
  private List<Transaction> transactions;
  …
}
```

More Rep Invariant Examples

```
class Polynomial {
   //Rep. invariant: degree = coeffs.length-1
   //  Poly is the sum of coeffs[i]*x^i for i ranging from 0 to
   //  degree
   private int degree;
   private int[] coeffs;
   // operations add, sub, mul, eval, etc.
}
```

```
class Polynomial {
   //Rep. invariant: for any 0 <= i,j < terms.size(), i != j =>
       terms_i.getExp() != terms_j.getExp()
   //  Poly is sum of terms…
   private List<Term> terms;
   // operations add, sub, mul, eval, etc.
}
```