CSE 203: ABSTRACT DATA TYPE

Dr. Mohammed Eunus Ali
Professor
CSE, BUET

Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

-- Linus Torvalds

Procedural and data abstractions

Procedural abstraction:

- Abstract from details of *procedures* (e.g., methods)
- Specification is the abstraction
- Satisfy the specification with an implementation

Data abstraction:

- Abstract from details of *data representation*
- Also a specification mechanism
  - A way of thinking about programs and design
- Standard terminology: Abstract Data Type, or ADT

The need for data abstractions (ADTs)

Organizing and manipulating data is pervasive

- Inventing and describing algorithms less common

Start your design by designing data structures

- How will relevant data be organized
- What operations will be permitted on the data by clients

Potential problems with choosing a data abstraction:

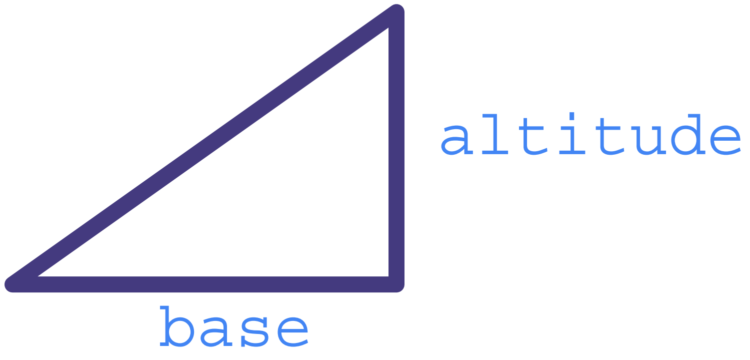
- Decisions about data structures often made too early
- Very hard to change key data structures (modularity!)

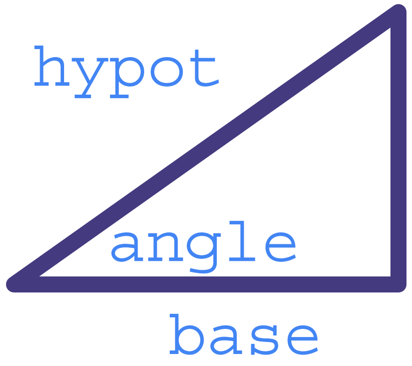
An ADT is a set of operations

- ADT abstracts from the *organization* to *meaning* of data
- ADT abstracts from *structure* to *use*
- Representation should not matter to the client
  - So hide it from the client

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

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





An ADT is a set of operations

- ADT abstracts from the *organization* to *meaning* of data
- ADT abstracts from *structure* to *use*
- Representation should not matter to the client
    - So hide it from the client

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

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

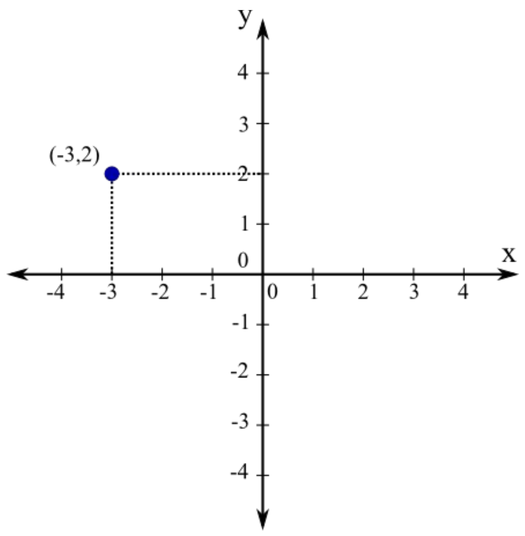
Instead, think of a type as a set of operations

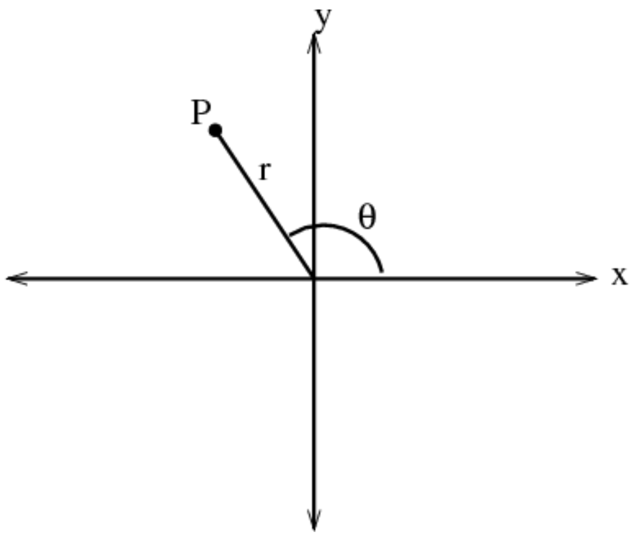
`create, getBase, getAltitude, getBottomAngle,` …

Force clients to use operations to access data

Are these classes the same?

```
class Point {      class Point {
  public float x;      public float r;
  public float y;      public float theta;
}           }
```





Are these classes the same?

```
class Point {     class Point {
  public float x;     public float r;
  public float y;     public float theta;
}          }
```

Different: cannot replace one with the other in a program

Same: both classes implement the concept “2-d point”

Goal of ADT methodology is to express the sameness:

- Clients depend only on the concept “2-d point”

Benefits of ADTs

If clients “respect” or “are forced to respect” data abstractions…

- For example, “it’s a 2-D point with these operations…”

- Can delay decisions on how ADT is implemented
- Can fix bugs by changing how ADT is implemented
- Can change algorithms
  - For performance
  - In general or in specialized situations
- …

Concept of 2-d point, as an ADT

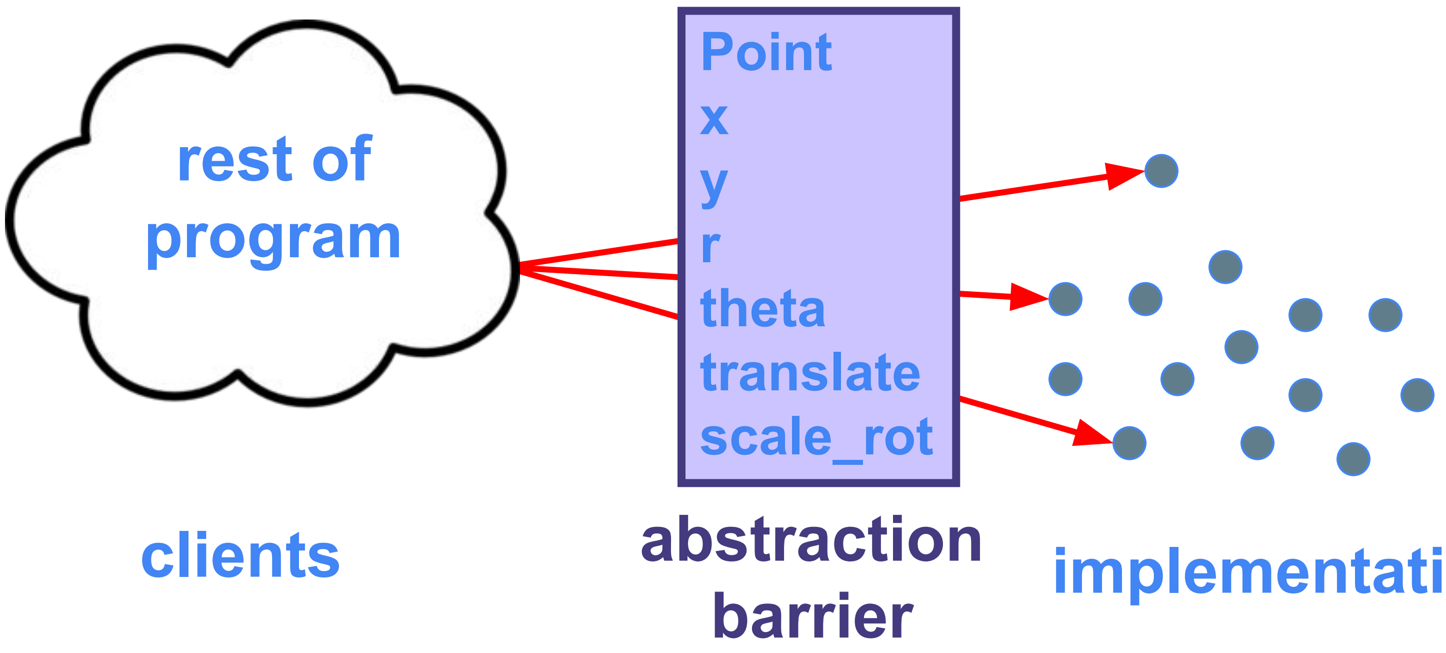
```
class Point {
  // A 2-d point exists in the plane, ...
  public float x();
  public float y();
  public float r();
  public float theta();

  // ... can be created, ...
  public Point(); // new point at (0,0)
  public Point centroid(Set<Point> points);

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

Observers

Creators/
Producers

Mutators

Abstract data type = objects + operations



Implementation is hidden

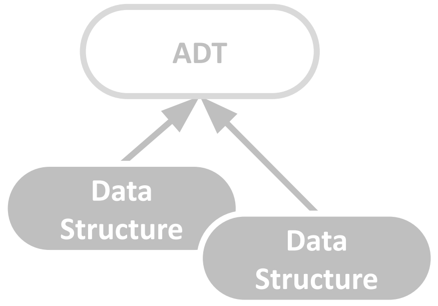
Only operations on objects of the type are provided by abstraction

Specifying an ADT

- Creators: return new ADT values (e.g., Java constructors)
- Producers: ADT operations that return new values
- Mutators: Modify a value of an ADT
- Observers: Return information about an ADT

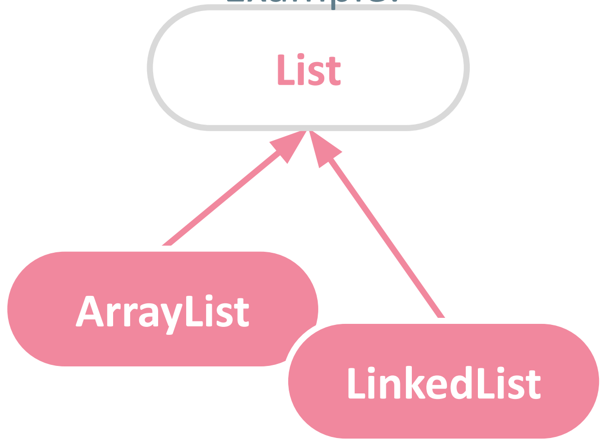
ADTs: Abstract Data Types

- An **abstract data type** is a data type that does not specify any one implementation.
  - Think of this as an agreement: about *what* is provided, but not *how*.



- **Data structures** implement ADTs.
  - **Resizable array** can implement List, Stack, Queue, Deque, PQ, etc.
  - **Linked nodes** can implement List, Stack, Queue, Deque, PQ, etc.

For Example:



Case Study: The List ADT

List: a collection storing an ordered sequence of elements.
-Each item is accessible by an index.
-A list has a variable size defined as the number of elements in the list
-Elements can be added to or removed from any position in the list

Relation to code/mental image of a list:

```
List<String> names = new ArrayList<>();  // []
names.size();                           // evaluates to 0
names.add(”Leona");                             // [“Leona”]
names.add(”Ryan");                         // [“Leona, Ryan”]
names.insert(”Paul”, 0);                    // [“Paul”, “Leona”, “Ryan”]
names.size();                           // evaluates to 3
```

Case Study: List Implementations

LIST ADT

State

Set of ordered items
Count of items

Behavior

get(index) return item at index
set(item, index) replace item at index
add(item) add item to end of list
insert(item, index) add item at index
delete(index) delete item at index
size() count of items

```
[88.6, 26.1,
94.4]
```

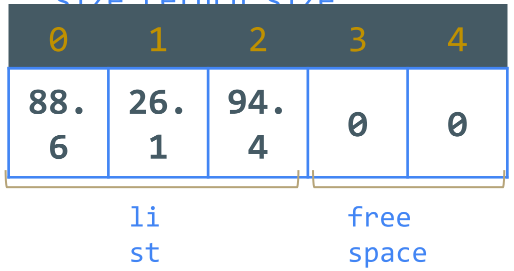
ArrayList<E>

State

```
data[]
size
```

Behavior

```
get return data[index]
set data[index] = value
add data[size] = value,
if out of space grow
data
insert shift values to
make hole at index,
data[index] = value, if
out of space grow data
delete shift following
values forward
size return size
```



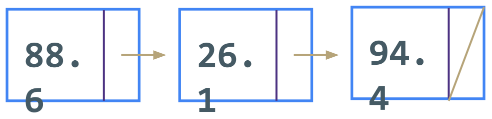
LinkedList<E>

State

```
Node front;
size
```

Behavior

```
get loop until index,
return node's value
set loop until index,
update node's value
add create new node,
update next of last node
insert create new node,
loop until index, update
next fields
delete loop until index,
skip node
size return size
```



Design Decisions

- For every ADT, many ways to implement
- Based on your situation you should consider:
  - Speed vs Memory Usage
  - Generic/Reusability vs Specific/Specialized
  - Robustness vs Performance

Design Decisions

- Both ArrayList and LinkedList have pros and cons, neither is strictly better than the other

- The Design Decision process:
  - **Evaluate** pros and cons
  - **Decide** on a design
  - **Defend** your design decision

- This is one of the major objectives of the course!

The End