Abstract Data Types

L.EIC

Algoritmos e Estruturas de Dados

2023/2024

P Diniz, AP Rocha,
A Costa, B Leite, F Ramos, J Pires, PH Diniz, V Silva

Procedural and Data abstractions

- Procedural abstraction:
  - Abstract from details of procedures
  - Specification is the abstraction
  - Satisfy the specification with an implementation
  - Use of a procedure depends on its purpose (what it does) but not on its implementation (how it does it)

- Data abstraction:
  - Abstract from details of data representation
  - Also a specification mechanism
    - A way of thinking about programs and design

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
  - Duplication of effort in creating derived data
  - Very hard to change key data structures (modularity!)

Abstract Data Type

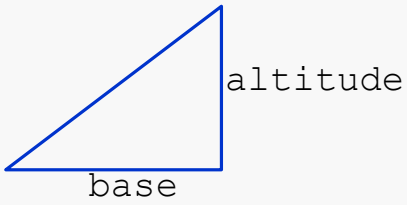
- ADT abstracts from the *organization* to *meaning* of data
- ADT abstracts from *structure* to *use*
- Representation should not matter to the user
  - So hide it from the user
- ADTs support *abstraction*, *encapsulation* and *information hiding*

Abstract Data Type = data + operations

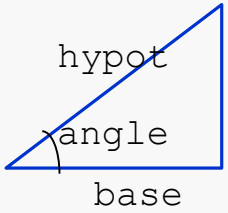
specification of data and operations that can be performed on the data (functionality)

Abstract Data Type

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




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



Representation should not matter to the user

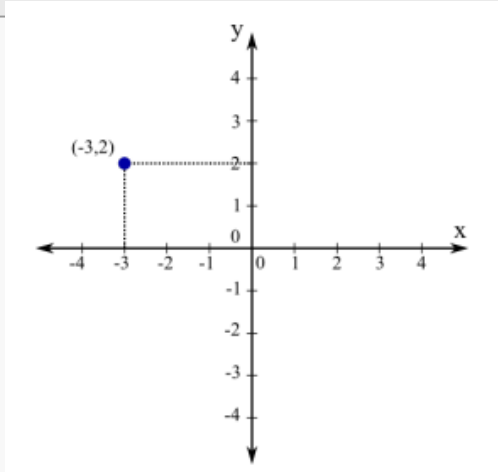
- So hide it from the user

- Instead, think of a type as a set of operations:

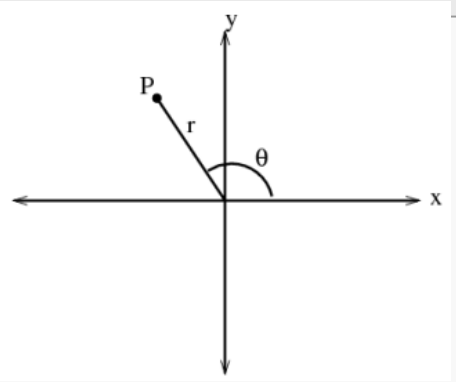
  create, ***getBase***, ***getAltitude***, ***getBottomAngle***, …
- Force users to call operations to access data

Abstract Data Type

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



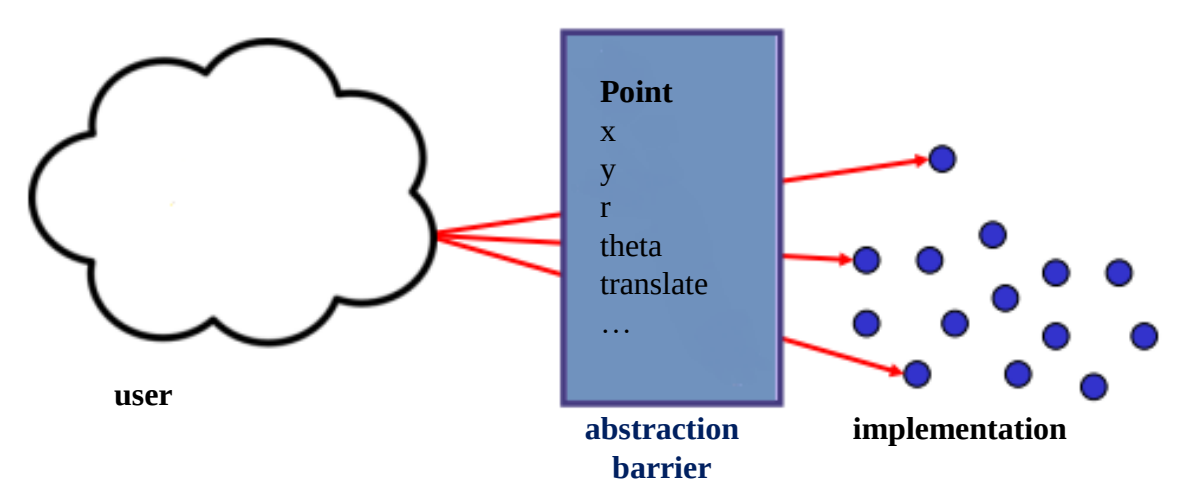

```
class Point {
   public float r;
   public float theta;
};
```



Are these classes the same?

- *different*: cannot replace one with the other in a program
- *same*: both implemente the concept of *"2-d point"*
- goal of ADT methodology is to express the sameness:
  - users depend only on the concept *"2-d point"*

Abstract Data Type



- Implementation is hidden
- Only operations on objects of the type are provided by abstraction

Abstract Data Type

- Specifying a data abstraction
  - A *collection* of procedural abstractions
    - not a collection of procedures
  - An *abstract state*
    - not the concrete representation in terms of fields, objects, …
  - Each operation described in terms of:
    - Creating
    - Observing
    - Producing
    - Mutating

Classifying types and operations

- Types (built-in or user-defined) can be:
  - **Mutable:** objects of a mutable type can be changed
  - **Immutable**

- Operations are classified as:
  - Creators: create new objects of the type
  - Producers: create new objects from old objects of the type
  - Observers: take objects of the abstract type and return objects of a different type
  - Mutators: modify objects

	mutable	*immutable*
creators	✓	✓
producers	✓	✓
observers	✓	✓
mutators	✓	✗

example: IntSet

```
// an IntSet is a mutable, set of non repeated integers.
// a typical IntSet is {x1, …, xn}
class IntSet {
```

- **Creators**
  - create a new object

```
// creates a new IntSet = {}
IntSet()
// creates a new IntSet = {x}
IntSet(int x)
```

example: IntSet

- **Observers**
  - take objects of the abstract type and return objects of a different type
  - used to obtain information about objects of the type

```
// returns true if x belongs to this, false otherwise
bool contains(int x) const

// returns the cardinality of this
int size() const
```

- **Producers**
  - operations on a type that create other objects of the type
  - more common in immutable ADTs, but mutable ADTs may have producers too

```
// returns the elements of this plus the elements of s2
// non-repeated (as a IntSet)
IntSet union(const IntSet &s2) const
```

example: IntSet

- **Mutators**
  - operations that modify an element of the type
  - not in immutable ADTs

```
// modifies this by inserting a new element x
bool add(int x)

// modify this by removing x
bool remove(int x)
```

Representation exposure

Representation exposure means that code outside the class can modify the representation directly

```
// immutable data type
class Tweet {
   string author,text;
   Date timestamp;
   Tweet(string a, string t, Date ts);
   string getAuthor();
   string getText();
   Date& getTimestamp();
};
```

intention: *Tweet* should be an immutable data type

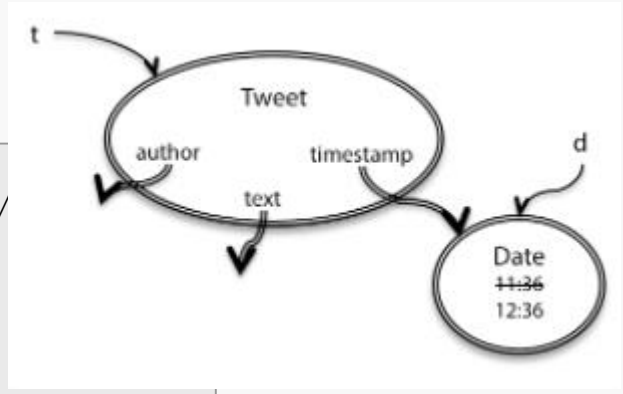
```
Tweet::Tweet(string a, string t, Date ts) {
   author = a; text = t;
   timestamp = ts;
}

Date& Tweet::getTimestamp() {
   return timestamp;
}
```

public Date getTimestamp() { return new Date(Date.getTime()); }

Representation exposure

Consider this user code that uses ***Tweet***:




```
// return a tweet that retweets t, one hour later*/
Tweet retweetLater(Tweet t) {
    Date d = t.getTimestamp();
    d.setHours(d.getHours()+1);
    return new Tweet("joao", t.getText(), d);
}
```

Problem: *Tweet* leaked out a reference to a mutable object that is immutability depended

```
// new version
Data& Tweet::getTimestamp() {
   Data *d = new Date(Date.getTime());
    return *d;
}
```

Lesson: storing a mutable object in an immutable collection can expose the representation